University of Otago

Masters Thesis

Enabling Real-Time Shared Environments on Mobile Head-Mounted Displays

---

Matthew G. Cook
Dunedin, New Zealand, March 2017

Thesis supervisors
Dr. Tobias Langlotz
Prof. Holger Regenbrecht
Dr. Steven Mills

To Kelly Jesse and Tobias Arthur Cook

Abstract

Head-Mounted Displays (HMDs) are becoming more prevalent consumer devices, allowing users to experience scenes and environments from a point of view naturally controlled by their movement. However there is limited application of this experiential paradigm to telecommunications — that is, where a HMD user can 'call' a mobile phone user and begin to look around in their environment. In this thesis we present a telepresence system for connecting mobile phone users with people wearing HMDs, allowing the HMD user to experience the environment of the mobile user in real-time. We developed an Android application that supports generating and transmitting high quality spherical panorama based environments in real-time, and a companion application for HMDs to view those environments live. This thesis focusses on the technical challenges involved with creating panoramic environments of sufficient quality to be suitable for viewing inside a HMD, given the constraints that arise from using mobile phones. We present computer vision techniques optimised for these constrained conditions, justifying the trade-offs made between speed and quality. We conclude by comparing our solution to conceptually similar past research along the metrics of computation speed and output quality.

Keywords. head-mounted displays, mobile computing, panoramas, video calls, computer vision, real-time

Acknowledgements

There are many who deserve thanks for supporting me throughout the research for this thesis. First among them my supervisors Dr. Tobias Langlotz, Prof. Holger Regenbrecht, and Dr. Steven Mills, who all provided valuable knowledge, support, and inspiration over the past year. Each were always persistent in their efforts in spite of my sometimes stubborn temperament.

I would like to thank the members of Otago HCI Lab, who are always available to bounce ideas off or simply for commiserations during the dragging stages of research. The Information Science department has also been generous with both time and material aid throughout my short research career.

Thank you to both of my parents, who always encouraged my curiosity about the world, and were ready to support my decisions from an early age. Finally I would like to thank my wife Kelly and son Tobias for their love and support throughout, and for being understanding of my many long days spent in front of a text editor.

Contents

List of Figures

Chapter 1

Introduction

Contents

Head-mounted displays (HMDs) have improved rapidly since the first prototype designed in 1968 [54]. Recent advances in display and tracking technology have enabled HMDs such as the Google Cardboard*, Samsung GearVR†, HTC Vive‡, and Oculus Rift§ to become desirable and increasingly common consumer devices. Users of these devices are able to have immersive experiences in virtual environments: 3D computer generated worlds rendered using a 3D graphics engine. Often however, these environments are based on real-world locations, created ahead of time by artists and software developers for users to explore. But what if we could virtually transport the user to a real, distant environment that is being both generated and updated in real-time, without the need for specialist equipment or knowledge? Achieving this would allow immersive experiences in arbitrary remote locations, it would let HMD users experience distant events live, facilitate remote assistance and virtual tourism, and enable a new form of creative interpersonal communication. By connecting a distant user with the HMD wearer in this way, the HMD user should feel as though they are really there with the other person. We believe that this

*https://vr.google.com/cardboard/

†https://www.samsung.com/global/galaxy/gear-vr/

‡https://www.vive.com/anz/

§https://www.oculus.com/rift/

'virtual transportation' will become one of the primary means of communication in the very near future. Therefore we aim to develop a novel application that will enable sharing remote environments with HMD users in real-time. We also intend to develop the application as a research platform — a base piece of software from which others can develop future work in related fields. This thesis will describe the concept and implementation details of the application alongside the issues and challenges encountered. We then evaluate its visual quality and computational performance characteristics.

1.1 Motivation

An important use case for video calling technology is to share your present environment with others — from mountaintop views to the interior of a new apartment. In these scenarios we do not simply want to convey information, we want our calling partner to feel that they are 'really there' in the environment, and for them to feel we share the space together. These two concepts are more formally called telepresence and copresence respectively [30], and simply *presence* when taken together.

A simple interaction model for sharing environments via video communication is for the party in the interesting environment (the sender) to point a phone camera out into their environment while the other participant (the receiver) views the video stream on a simple screen such as a mobile phone or computer monitor. Should the area of interest be greater than the field of view of the camera, the sender will move and rotate the camera to show more of the scene. Whilst this provides more information to the receiver, their point of view is completely out of their control. To examine particular details in the environment the receiver must request that the sender stop moving the camera, or perhaps revisit an area they have already moved on from. Lombard and Ditton [34] theorised that by mediating the interaction in this fashion users would have a lower sense of presence in the remote environment; Müller et al. [40] confirmed this to be the case by developing and testing a phone-to-phone prototype called PanoVC where the receiver was able to control their viewpoint.

Compared to viewing an environment on a simple screen, we expect a head-mounted display to provide a more immersive experience, and therefore a stronger sense of presence, for the receiving user. However, we cannot simply display the video stream controlled by the sender as in the simple interaction scenario described above — the conflicting motion cues between the display and the user's own movement are extremely likely to cause motion sickness symptoms [21]. Because of this we must use a system that allows

the receiver to control their own viewpoint. To develop such a system we will borrow ideas and employ lessons learned from the PanoVC project, however we will demonstrate that our implementation must be quite different to achieve the demanding performance requirements of HMDs.

1.2 Immersive Telepresence System

Our goal is to work towards *ubiquitous immersive telepresence*. Ubiquity is provided by mobile phones — nearly everyone has one and they can be taken anywhere, allowing for almost any location to be shared without needing forward planning. Head-mounted displays provide the immersion by blocking out a user's current environment and allowing them only to see and hear the remote environment. Telepresence will be supported by the software we build, which allows the HMD user to control their viewpoint in a high-quality, continuously updating, real-time representation of the remote environment.

We propose – based on prior research [6] [34] – that a system which allows a user to explore the remote environment using a HMD will provide a greater sense of immersion, telepresence and copresence. The key research questions are therefore: is it possible to develop an application that can transport an HMD user to any remote environment with just a mobile phone? And will it be able to generate and transmit the remote environment with sufficient speed and quality to enable an enjoyable telepresence experience? We will attempt to achieve this by utilising the existing onboard sensors of mobile devices, along with novel use of computer vision to generate high-quality, real-time updating environments.

By specifically targeting mobile phones for environment generation we grant an additional degree of relevance to the research; upon completion nearly anyone on Earth can immediately begin to share their environment in real-time to distant parties. Due to this decision however, we find ourselves working in an incredibly resource constrained environment. For that reason this thesis focusses on the technical aspects of development, with the aim to demonstrate that immersive environment generation and sharing is possible on current mid-range to high-end mobile phones. Thorough evaluation of the effects of the system on telepresence and copresence is therefore left to future research.

The use of mobile HMDs – wireless devices with their processing power and display provided by an inserted mobile phone – presents further challenges for our implementation. The performance requirements for any given mobile HMD are equal to the maximum possible display output for the attached phone. In this research we have used a Samsung

GearVR HMD with an inserted Samsung Galaxy S7[¶], which has a maximum resolution of 2560×1440 pixels and a refresh rate of 60Hz, which will be challenging to achieve.

We also intend for our application to serve as a platform for further telepresence research in the future. This dictates a modular approach to our application design so individual components can be replaced for testing and comparison. This also implies there is no upper limit on performance that we should aim to achieve — less computational resource usage in our application means more left over for future novel ideas.

1.3 Shared Panoramic Environments

Figure 1.1: Overview of an immersive panoramic call in progress. The sender (left) builds a panorama (centre top) by rotating their phone, with their current view shown in red. The receiver (right) can view any part of the environment for which the panorama has been built — their current view is shown in blue.

We have decided to realise our telepresence system with what we call *immersive panoramic calls*. The remote mobile phone user remains in a fixed location while they film their environment with the rear (outward) facing camera. Using a combination of the onboard sensors and computer vision algorithms our system will map the recorded video frames into a continually updated spherical panorama in real-time. The panoramic map is shared via the internet with the HMD user who is able to explore the environment

[¶] `https://www.samsung.com/us/explore/galaxy-s7-features-and-specs/`

with natural movements, i.e., by rotating their head. As the environment representation is updated in real-time, if both users are looking in the same direction the view of the HMD user will appear as a live video, but without the risk of motion disassociation if one of the users turns away. Figure 1.1 shows an overview of the system in use, with the users observing slightly offset views of the environment from one another.

A panoramic map is a 2-dimensional representation of a 3-dimensional environment created by projecting views of the 3D space onto the map. Creating such a map is analogous to creating a map of the Earth from a globe, however in our case we observe the environment from the inside looking out. Because of this projection step panoramas are most suitable for environments with distant features, primarily outdoors. The simple computer storage format of panoramic maps also enables a straightforward method of updating the environment in real-time — instead of a more traditional approach of only adding information to parts of the panorama that have not yet been mapped, we overwrite all areas for which new information arrives, creating a dynamic panorama that is updated live as the environment changes.

To speed the development of our prototype application we use a number of free and open source libraries throughout this project to avoid working on previously solved problems. We use the Android Software Development Kit (SDK) and Native Development Kit (NDK) for access to device sensors and cameras, WebRTC for real-time communication of video, audio and data over the internet, and OpenCV for its collection of computer vision algorithms.

1.4 Results and Contribution

We successfully developed a novel system for connecting mobile phone and HMD users using an immersive panoramic call. It is backed by a computer vision (CV) implementation that balances robustness and performance to create high resolution panoramas shared over a wireless network on the fly. The overall system design and the specific CV algorithm form our primary contributions. Whilst we fell slightly short of our own performance goals, our system reached speeds that are considered real-time in mixed reality applications, and it had a low end-to-end latency. In informal user testing, subjects have felt a sense of presence in the remote environment.

We also have reached our secondary goal of developing a research platform for future work, as demonstrated by a PhD student already integrating his work on gestural user interfaces into our system. During work on our implementation we also uncovered errors

in OpenCV – a computer vision library with over 14 million downloads [44] – and have since had a patch accepted and released fixing the issue. We have also prepared a module for the WebRTC project that will be suitable for submission after the conclusion of this work.

Overall we argue that the work presented in this thesis provides a solid foundation for a new method of communication with potential for both research and practical applications in the near future.

1.5 Thesis Outline

- **Related Work (Chapter 2):** We cover relevant past research in the areas of virtual and augmented reality, head-mounted displays, image stitching, panoramas, and video communication. We pay specific attention to the technical aspects of related projects.
- **Conceptual Framework (Chapter 3):** We provide an overview of the concepts and techniques required to realise our application.
- **Implementation (Chapter 4):** This chapter details the software implementation of our environment sharing application.
- **System Evaluation (Chapter 5):** We show quantitative results regarding the performance and quality characteristics of our system.
- **Summary and Conclusion (Chapter 6):** We provide a summary of our output and findings and propose future work made possible by our developed platform.

Chapter 2

Related Work

Contents

This thesis draws upon work from a variety of fields including mobile human-computer interaction (HCI), HMD technology, telepresence, computer vision and software engineering. In this chapter we highlight papers that similarly combine facets of these fields, or provide an overview of the state of the art in their field. We also give special consideration to the key papers [40] [60] detailing the implementation of the PanoVC environment sharing software, as we use their benchmark results as a point of comparison to our own implementation.

2.1 Presence

One of the key goals of the work in this thesis is to develop a platform that will enable research into the elements affecting presence for users in a real-time shared environment. As such we must first examine research that helps to define presence and its constituent factors such as telepresence, spatial presence, and copresence, along with how these factors are affected by system inputs (hardware, software, and interaction) and how they affect user experiences such as enjoyment and satisfaction.

Lombard and Ditton [34] provide an extensive overview of presence related research to that point in time (1997), covering many of terms of presence we are concerned with, though it is not considered a formal taxonomy. Presence itself is defined as the perception of any mediated experience that feels as though it is non-mediated. To this end any form of media can induce a feeling of presence, though each media form is inherently likely to do so to a lesser (novels), moderate (cinema), or greater (virtual reality) degree. Importantly for our work, they note the distinction between the terms *telepresence* and *virtual presence*, defined as "feeling like you are actually 'there' at the remote site" and "feeling like you are present in the environment generated by the computer" respectively. Even though we use virtual reality (VR) devices and a computer mediated experience, we aim to achieve telepresence, as our represented environments are all real world locations.

Though [34] discusses the concept of presence with regards to sharing a space with others, it was yet to be formally defined. Durlach and Slater [13] explore this idea, *copresence*, stating that participants must already have a strong sense of presence in a common virtual environment, they must be able to communicate, and they must perceive that the environment is consistent between them, e.g. by talking about a common visual feature, or the interaction of one user affecting the environment of all users. A more recent (2004) overview of presence topics [30] discusses copresence in contrast to *social presence* — where the latter requires primarily that people can communicate with and understand a range of social cues, copresence requires that "individuals become accessible, available, and subject to one another", a more demanding set of requirements building upon those for social presence.

A related concept to presence is that of *immersion*. In colloquial speech the term is often used to describe the feeling of presence as defined above. However research literature [6] defines it much more narrowly: "immersive is a term that refers to the degree to which a virtual environment submerges the perceptual system of the user in computer-generated stimuli. The more the system captivates the senses and blocks out stimuli from the physical world, the more the system is considered immersive". Achieving immersion in VR has since been shown to improve the feeling of presence [51] [63], so these targets of submerging the perceptual system and blocking external stimuli are something we should strive for.

The Biocca and Delaney article [6] that provided the preceding definition also explains the relationship between immersion and virtual reality technology, although it is very much a product of its time. They describe existing VR implementations as "[like a] prototype,

a jumble of wires, LCDs and artful technical compromises", and describe a display with 440×250 pixels as "medium resolution". Fortunately, with the availability of consumer HMDs with over 30 times the display resolution these statements have become outdated. Many statements remain as true now as in 1995 however, and can be used to guide our development. They state that a virtual reality system should act as an extension of our senses, and that can only be achieved by delivering stimuli in a natural way. They also describe the properties of the ideal VR HMD, which include filling the visual field, matching the resolving power of eyesight, and coordinating visual-spatial cues. Whilst some of these goals are aided by modern hardware, it must also be supported by the software we write.

Schubert et al. [51] performed two survey based studies and corresponding factor analyses which aimed to determine the disjoint experiential components that combine together into a presence experience. They found that the key factors leading to the experience of presence are spatial-constructive mental processes, interaction and attention, and the judgement of realness. The key takeaway for the development of high presence systems is ensuring these mental processes are not blocked by incompatible sensory stimulus.

Drugge et al. [12] have created a telepresence system for the purpose of evaluating the effect of wearable computers on both telepresence and remote interaction. The system involved the use of a monocular head-mounted display, a head-mounted camera and portable keyboard which together allowed users to participate in a teleconference. This system was compared to conference participation via a desktop computer, and whilst only evaluated qualitatively and normatively, was found to improve the sense of telepresence in the case of wearable computers. They also focused on questions regarding future applications, in particular how to improve the overall enjoyability of experience, and how to aid with deployment and user adoption. They recommend that applications must have a low time to setup before use and that any equipment that is not represented in the remote environment should not be actively interacted with.

To determine the effects of interrelation between social and spatial presence Horvath and Lombard [20] created a 2×2 study using a menu driven application with four designs: the first being a basic web menu interface; one designed for social presence by addition of a personable avatar to communicate with; one designed for spatial presence by assigning functionality spatially to buildings; and the last being a combination of social and spatial. After participants had used the software to achieve assigned goals, they took surveys first to determine if the desired presence factors had been invoked, then to assess attitudes on a

number of factors: overall satisfaction, enjoyment, comprehension, perceived user ability, and likelihood to use the application. In every case except for 'overall satisfaction' with regards to social presence, a sense of social or spatial presence was found to enhance the experiential factors. Additionally, they found that combining social and spatial presence together had a stronger effect in all cases than either in isolation. This result confirms that achieving these presence factors will be desirable for our application.

Rae et al. present a paper titled *A Framework for Understanding and Designing Telepresence* [47]. By reviewing prior research, conducting surveys, and conducting a field study of two prototype designs, they developed a framework consisting of 7 design dimensions and 17 scenario categories which can be used to inform design decisions when creating new telepresence systems. Pece et al. [46] provide further design guidance by comparing the experience of using different devices to complete an observation task in a panoramic environment. Importantly for us they found that whilst HMDs were tied for top performance in task completion, they scored poorly for usability (Rank C in System Usability Scale (SUS) classification [33]). They speculate that this may have to do with their users' unfamiliarity with head-mounted devices.

Taken together, the papers in this section provide definitions for key ideas related to our research. *Presence* is the perception that a mediated experience feels as though it is not actually mediated. *Telepresence* is the sense of being transported to a remote location. *Copresence* requires mutual awareness of people in a shared space. *Immersion* is the extent to which external stimulus is diminished and computer-generated stimulus is engaged. In addition to these definitions the research shows evidence of presence factors on user enjoyment and usability, and techniques that can be used to increase the perception of presence.

2.2 Shared Environments and Telepresence

There has been much research into the topic of shared virtual environments wherein multiple users of a computer system experience copresence with each other, i.e., they feel that they share the virtual environment together. We can narrow this field further by selecting systems where the 'virtual environment' is in fact only virtual for a subset of participants, or in other words there is a real, non-computer generated environment which some parties can perceive directly. The papers in this section therefore all combine elements of shared virtual environments derived from real environments, copresence, and telepresence — a property shared with our proposed system.

An early entry into shared environments with a user controlled point of view is the teleconferencing project by Yang et al. [63]. They created a telepresence system for group teleconferences using a technique based on light field rendering. This would ordinarily involve hundreds of cameras, however they assumed that user's eye height would be constant and that their positional change would be minimal due to fixed seating arrangements, allowing the use of only 11 cameras per group. The system allowed two groups of users in separate environments the feel as though they were across a table from one another via accurate stereoscopic rendering of the opposite group from each user's point of view, determined in real-time. Bi-directional capture, rendering, and communication is supported. The system was evaluated quantitatively for its performance characteristics – and found to be running at a low 8 frames per second (FPS) – and qualitatively for its visual accuracy.

Continuing with teleconference applications, Pece et al. developed PanoInserts [45], a system where multiple live video streams are registered and stitched into a common, precreated panoramic scene using both marker and image-based computer vision techniques. The live videos are provided by mobile phones, but because they perform no other processing than streaming video, they could equally be replaced by webcams or other simple cameras. The static scene panorama itself is captured with multiple images from a standard handheld digital camera and stitched with the PTGui software package. Once the static panorama and dynamic 'inserts' are merged, the final panorama can then be viewed on a desktop computer by a third-party. The system was evaluated with a virtual object placement task and an SUS survey — they found that compared to a standard direct webcam call PanoInserts gave higher task accuracy, but had no effect on task completion time. For usability PanoInserts scored a Rank B, compared to the webcam system's Rank A.

Jo and Hwang [23] present Chili, a phone-based system that allows a remote user to explore the environment around a local user in real-time. They achieved this by designing an application where phone rotations performed by the remote user are transmitted as commands instructing the local user to match that movement. Though they did not perform a user study on presence factors, we predict this human-mediated view control trades away the feeling of telepresence in exchange for stronger social presence and copresence. Regarding performance, the application transmits video at 15 FPS.

Gauglitz et al. [16] developed the first prototype telepresence system leveraging the technique known as *simultaneous localization and mapping* (SLAM). This involves a video feed from a camera being used to create 3D representation of the scene in real-time whilst

simultaneously using that model to determine the camera's position. SLAM is performed on a tablet device controlled by a local user, while the environment model is transmitted in real-time to a remote user who is able to explore the scene using a desktop computer interface. The remote user can also add annotations spatially registered in the scene, which are shared between both users. The system was tested with a remote expert-local worker user study and compared against a simple video call with static screen annotations; whilst the new system was not found to significantly affect the task completion speed or accuracy, participants strongly preferred the SLAM based interface.

The Polly system by Kratz et al. [28] allows for viewport control on a direct physical level — they mounted a mobile phone inside a motorised three-axis gimbal and allow for the rotation to be controlled by a remote user using a desktop interface. The gimbal can be placed statically on a tripod, or mounted on a harness near a calling partner's shoulder (like a parrot – hence Polly). Video and audio communication was provided by existing video calling software such as Skype*. A preliminary qualitative evaluation found that user enjoyment gains are actually most likely for the local user, as they can now act as a guide whilst still having their hands available for other tasks.

Over a series of three papers [25] [26] [27] Kasahara et al. present the systems JackIn and JackIn Head. The former is a system where a remote user wears a head-mounted display and head-mounted camera; by using SLAM techniques the streamed video frames are assembled into a 3D environment representation which a remote user can explore with a desktop computer interface. The remote user can also point into the scene, detected with a gestural input device (Leap Motion†), which will create spatially registered annotations in the view of the local user. JackIn Head extends the system by using a head-mounted omnidirectional camera on the local user and allowing the remote user to explore the scene using an HMD. When this system is used in a real-time mode – where the local and remote users' positions are synchronised and only rotation is controlled by the remote user – the researchers found it is imperative to have both hardware and software based image stabilisation to avoid inducing motion sickness.

These papers showing applied research into novel telepresence systems have a similar structure: based on either previous research or capabilities of emerging technology a system is developed and evaluated. Depending of the complexity of the underlying system, these evaluations tend to be either on performance and correctness or presence and usability. Even though this was a small sample of available research we can see a trend emerging

*`https://www.skype.com`
†`https://www.leapmotion.com/`

in the systems; first they transport whole rooms, then individuals at a desk, and finally people outdoors using mobile or wearable computers. This final category is of ongoing research interest today, especially when combined with the ubiquitous hardware carried by nearly everyone – mobile phones.

2.3 Panoramic Environment Representation

When creating and showing virtual environments there are some key decisions to be made about how they are generated and rendered. First, is the environment real or virtual? For real environments you must then decide how to store the environment that is captured: point cloud, 3D geometry with textures, or projected map (panorama)? In this section we review systems using a panoramic representation of a real environment.

Neumann et al. [42] present a system for capturing panoramic video streams designed for later playback in an HMD. They used a system of 5 cameras looking at a 5-facet pyramid mirror to capture a 360 degree scene simultaneously, excluding the sky and ground. The panorama is stored as separate video files along with the camera parameters as determined ahead of time in a separate calibration step. When stitched together, the panoramic videos provide a total resolution of 3000×480 over 360 degrees. There is no computer vision required for stitching — instead the streams are decoded simultaneously during playback and subsections of one or two videos are shown to the viewer based on their head pose as determined by the inertial-magnetic sensor inside the HMD. The system was used in a pilot qualitative study and judged to provide "a strong sense of immersion". This system from 2000 is one the earliest examples of recording a 360 degree panoramic scene, following on from Nayar's work in 1997 [41], and when combined with HMD playback forms the basis of virtual reality video which are just becoming popular today, for example via services like YouTube360[‡].

In contrast to panoramic videos which run once and come to an end, Agarwala et al. [2] present *panoramic video textures*, which are panoramas with a mix of static and dynamic areas. The dynamic areas show motion looping infinitely and thus are most suitable for features with endless motion but without any change to their overall position, for example water flowing over a waterfall. They designed the system to enable capturing of panoramas without any special equipment – any standard video camera will work – though they require a particular motion model to be followed for accurate results. Once a video of the scene is captured the video texture is generated in an offline process using

[‡]`https://www.youtube.com/channel/UCzuqhhs6NWbgTzMuM09WKDQ/about`

feature matching computer vision techniques; first requiring manual input for identifying dynamic areas. The resulting panoramas can cover up to 180 degrees and have a vertical resolution of 720 pixels. The system was evaluated qualitatively with results described as "compelling and immersive", tough "careful examination does reveal the occasional artifact". One of the main issues identified was the performance of the offline generation — with 2 minutes of video footage it would take between 2 and 7 hours to create the final panorama.

Tang et al. [58] developed an immersive telepresence system dubbed *immersive cockpit* with design goals of being "real-time, live, low-cost, and scalable". The system records a 150 degree scene from 8 cameras and presents in real-time the stitched panorama to users via a hemispherical display that encompasses their field of vision. Stitching parameters are determined once during initialisation of the system using feature based calibration and then remain constant during run time. The research focusses on the constraints caused by handling 8 camera feeds at once in real-time (e.g. compression and networking) and also on finding the "sweet spot" for viewing a hemispherical screen such that the viewpoint is spatially accurate. The system was evaluated through visual inspection and performance profiling, which showed it running at 20 FPS.

Recognising the desire to create panoramas for augmented reality applications 'on the fly' without any special equipment, DiVerdi et al. [11] developed Envisor, a system that aids a user in capturing their environment for representation as a cubed-mapped panorama. The system is designed for use with either a handheld or head-mounted camera and performs frame-to-frame tracking and mapping using the video stream. The computer vision techniques employed are Shi-Tomasi corner detection [52] followed by Lucas-Kanade optical flow tracking [36], which is a method evaluated (and eventually rejected) for our own system (see Section 4.8.3). They also employed SURF feature descriptors [5] to create landmark points to reduce rotation drift. The system was evaluated with a visual inspection of the resulting cube-maps and performance measurement. They found an acceptable panorama could be generated at 15 FPS on contemporary (2008) desktop workstations.

Dalvandi et al. [10] performed a study to determine the effect of using panoramic video on the overall sense of presence. The study compared conditions of 'panoramic video', 'regular video', and 'slide show of panoramic still images' to display an environment in which a user must navigate a route and recall landmarks. The panoramic video was acquired ahead of time with 8 cameras operating at 422×316 resolution and 10 FPS. The

study had a within-participant design and was evaluated with a presence questionnaire, finding that the "sense of presence was highest in panoramic video condition" in spite of the low resolution. Participants commented that because they were moving along a route while looking around a dynamic panorama they could become disoriented quite easily.

Whilst the fundamentals of storing and rendering a panorama are a solved problem, the above papers show that both the performant generation of panoramas and how best to use them in support of telepresence systems remains an area of ongoing research. There is also interest in the use of non-specialised equipment for capturing panoramas to allow augmented or shared environments at any location.

2.4 Real-time Image Stitching

A consistent feature of the works in the preceding sections is that there has been little research regarding the real-time creation of panoramas from a single, non-wide angle camera. Many systems that do not employ fish-eye or 360 degree cameras build their environment in an offline processing step ranging in duration from minutes to hours; those that have real-time stitching do so via multiple cameras with fixed positions that are calibrated ahead of time. In this section we review work regarding techniques of panorama creation, in particular how they can be applied to live streams of video such that creation occurs in real-time.

Szeliski [55] introduces the concept of *video mosaics* which are a wide-angle planar view of a scene created by stitching consecutive video frames, on the condition that a particular camera motion pattern is followed. Compared to preceding image stitching methods video mosaics take advantage of the large overlap in content between adjacent frames. Szeliski and Shum [57] later generalised the technique to allow creation of omnidirectional panoramas from arbitrary camera rotations about a fixed position. In 2006 Szeliski included these findings in *Image Alignment and Stitching: A Tutorial* [56], a summary of the state of the art in computer vision stitching techniques, which remains the canonical reference on the topic.

When creating panoramas from videos it is likely that elements in the scene are moving independently from the camera, e.g. swaying tree branches. To counteract the potential misalignment from this Fitzgibbon [15] introduced the concept of *stochastic rigidity* to model camera poses distinctly from dynamic elements. Whilst this technique would be especially applicable to real-time panorama generation, the implementation provided was far too computationally expensive to be tested in that scenario.

Baudisch et al. [4] present Panoramic Viewfinder, a system for creating low resolution previews of panoramas stitched from individual still images. The system is intended to guide users taking pictures for panoramas and thus provides real-time feedback of the location in a panorama the camera's current view would be projected to. Due to this design only half the work of panorama generation needed to be solved in real-time – alignment but not projection. Even with the reduced workload the low computation power of hand-held devices in 2005 required that alignment was performed on a downsampled, low resolution copy of the viewfinder image. As with other computationally intensive prototypes this work was evaluated only in terms of accuracy and performance; it was found to provide a useful preview of stitching location as the viewfinder updated in real-time.

Adams et al. [1] created a general optical tracking system for mobile devices, listing panorama creation assistance as one of the possible applications for the technique. They perform frame-to-frame camera position and rotation tracking by performing edge and corner detection and alignment on consecutive frames acquired from a mobile phone camera. This algorithm performs best for scenes without recurring patterns and was shown to have higher accuracy outdoors than indoors. The system ran at 30 FPS when using a camera input stream resolution of 320×240.

One of the most recent and most relevant works in the field of real-time panoramas is that by Wagner et al. [60] who developed a method to create accurate panoramas in real-time on mobile phones. Rather than performing frame-to-frame tracking of camera positions, the software compares the current camera frame to the existing built panorama and fills in any new area that was detected. The created panorama can represent the full 360° horizontally but only 76° vertically. The computer vision techniques employed by the system are FAST corner detection [49] followed by template matching using normalised cross-correlation [32]. This method resulted in a system that runs in real-time (30 FPS) on a camera input of 320×240 with an output panorama size of 2048×512.

Following on from this work, Müller et al. [40] present PanoVC, a telepresence system where two distant mobile phone users can share or exchange a panorama of their environment. The underlying real-time panorama generation is built directly from the methods and source code of the Wagner [60] implementation above, but modified such that instead of simply filling new areas of the panorama as they come into view, all new vision is incorporated, making it more like a video panorama. The panoramic data is sent to a connected partner who may then look around at the shared environment by rotating their own phone. A user study showed use of PanoVC improved both spatial and social

presence when compared to a standard mobile phone video call. In spite of the hardware improvements since Wagner's initial work the system does not run any faster due to the overhead introduced by sharing an environment between devices.

2.5 Summary

The research in this thesis touches on many topics including presence, shared environments, panoramas, and real-time computer vision. We have outlined some of the key papers and developments in the respective fields above, although they are a small sample of what are active and ongoing fields of research.

From the research into presence we derive useful terms for describing the feeling of 'being there', and the research shows how this feeling translates to a more enjoyable experience. A series of novel systems papers provide a framework for evaluating our developed prototype. Finally, the papers developing or utilising computer vision techniques for panorama generation can guide our implementation approach.

With regards to our proposed system there are still many unanswered questions, first among them the overall feasibility on today's devices. Existing implementations of computer vision on phones show the difficulty of achieving real-time performance even at low resolutions. Before we can find out what effect real-time updates of a shared environment have on an HMD user, we must first develop an implementation of panorama generation that can support the high resolution and frame-rates demanded by HMDs. Following the example of papers described in Section 2.2 we will implement and evaluate our system for performance and accuracy in this thesis, leaving a usability and presence study for future work.

Chapter 3

Conceptual Framework

Contents

Our novel system will enable the real-time sharing of environments through *immersive panoramic calls*: a way of sharing the view of everything around your current location. Users should have the feeling of being virtually transported to a remote location – as it appears in the current moment – and they should feel present with a calling partner in that environment. The user at the remote location should require no more equipment than a mobile phone and our software. To enable this feeling of transportation we combine mobile head-mounted displays with software for creating and viewing real-time spherical panoramas.

In this chapter we will examine the process of making and participating in an immersive panoramic call, the required hardware and software, and the mathematical and technical concepts that enable the experience. We will also give an overview of the data flows and transformations throughout the call.

3.1 Hardware Platform

One of the elements that allows us to explore the topic of sharing arbitrary remote locations is the array of onboard sensors that are now standard on most Android devices. To realise our implementation we depend on three separate categories of sensors: video, audio, and positional sensors. Conversely, the output capabilities our devices, in particular the display used inside of the mobile HMD, place restrictions on our application by defining the target quality we must reach.

3.1.1 Mobile Phones

Before discussing the specifics of phone hardware, the first choice to be made is that of which operating system (OS) to develop for; there are two sufficiently ubiquitous options: Google's Android and Apple's iOS. We chose the Android OS for our development platform as iOS presently has no mobile HMD support. Whilst there is no reason in principle why the mobile user could not use an iOS device to send to a receiving Android based mobile HMD, having both call participants running the same operating system makes for much simpler development in the prototype stage.

For our development and evaluation we targeted two specific Android devices, the LG Nexus 5* and the Samsung Galaxy S7†. These two devices give a useful spread of performance characteristics; they were each considered high performance at time of release though the former was released in 2013 and the latter in 2016. The effect of the performance differences between these devices is detailed in the evaluation chapter. As an additional requirement, we need a phone that is compatible with the Samsung GearVR HMD, which the S7 fulfils.

Our target phones are each equipped with a front and rear camera — to acquire vision of the environment we only need access to the rear-facing camera. To create panoramas without discontinuities caused by automatic control of the camera's focus, exposure, and white balance we must access the camera with a newer set of APIs called `camera2` ‡ that allow manual control of these features. As these APIs are only available in Android 5.0 and later – which shipped after the release of the Nexus 5 – the phones need to be upgraded to at least this version of Android.

Besides cameras, we also have access to a set of sensors that provide information

*`http://www.lg.com/us/cell-phones/lg-D820-Sprint-Black-nexus-5`
†`https://www.samsung.com/us/explore/galaxy-s7-features-and-specs/`
‡`https://developer.android.com/reference/android/hardware/camera2/package-summary.html`

Figure 3.1: The mobile phones we used to develop and test our prototype, the Nexus 5 (left) and Galaxy S7.

about the current position and rotation of the phone. The Android sensors specification [17] provides a complete list of sensors which may be present on phone. Those we are interested in – and which are present on both of our test devices – include the gravity sensor, gyroscope, and geomagnetic field sensor. Taken together these sensors can be combined to give device rotation in the context of an absolute frame of reference — in fact, this functionality is already provided by the OS as a 'virtual sensor' called the *rotation vector*. As our panoramas are not sensitive to small changes in position we can ignore those that deal in position only such as the accelerometer and GPS sensor.

3.1.2 Calibration

Whilst nearly all functionality of our phones can be used 'out of the box', we required one additional piece of information derived from a calibration step before we can proceed: the intrinsic parameters of each phone's camera. These parameters are the focal length, principal point, and distortion coefficients. Intrinsic parameters are constant for a given camera and therefore only have to be calibrated once, which can be contrasted to the extrinsic parameters of position and rotation that will be changing while the camera is in use.

Newer Android phones provide the `LENS_INTRINSIC_CALIBRATION` API to acquire

these parameters from factory calibration — unfortunately neither of our phones support this so we must calibrate manually ourselves. We use a technique developed by Zhang [64] whereby we can estimate the intrinsic parameters from multiple distinct views of an easily detected planar pattern, in our case a chequerboard. Figure 3.2 shows a subset of the views taken, with pattern detection overlaid; for an accurate calibration we used 80 such views for each camera.

Figure 3.2: Three of the images taken during calibration of the Galaxy S7. The coloured overlay shows computer vision recognition of the chequerboard pattern.

3.1.3 Head-Mounted Displays

Figure 3.3: Samsung GearVR, the mobile HMD used in our prototype, with Samsung Galaxy S7 fitted as the display.

Consumer HMDs can be broadly divided into two main categories: PC driven, wired

devices that track both translation and rotation in a limited area, such the HTC Vive and Oculus Rift, versus mobile processor driven, wireless devices that can be used anywhere but only track rotation. As our panoramic environments appear the same to a viewer regardless of their position we are not forced into either option, however we have chosen to develop for the GearVR, which belongs to the latter category. This decision was primarily driven by potential future research — for example our application could be adapted to target two GearVRs such that two users could swap environments in real-time. Another motivating factor is that mobile HMDs are far cheaper than their PC powered counterparts, which supports our goal of creating a relevant, universally applicable solution.

The GearVR must use a compatible phone for both its display and processing power; in our system we use a Galaxy S7. To create a compelling HMD experience we must match the native display resolution and frequency of the inserted phone, which for the S7 is 2560×1440 pixels at 60Hz — a significant target. Although the phone has its own rotation sensing capabilities as described in the preceding section, HMDs demand a higher level of precision than the phone can provide in order to avoid motion sickness. Therefore the GearVR has an onboard inertial measurement unit (IMU) [29] which provides rotation updates to the phone via USB.

3.1.4 Alternative Devices

One possible option for the environment sharing device was to use a dedicated panoramic camera, for example the Ricoh Theta[§]. As these devices are able to send complete 360 degree frames in a video stream they have a significant advantage in ease of implementation. There are three factors working against these devices however: the panoramic video frame has a comparatively low angular resolution compared to full frames from a narrower field-of-view camera; the devices are not nearly as commonplace, reducing the practical relevance of the research; and without the ability to run arbitrary programs, they are a less useful platform for future research that may incorporate user interactions, including bi-directional communication. For these reasons we ruled out panoramic cameras in favour of mobile phones.

[§]https://theta360.com/en/

Figure 3.4: Ricoh Theta (left) and an example 360° panorama taken by it in equirectangular projection.

3.2 Communication Protocol

As our system is reliant on network communication, we must use or develop a protocol so that participants can both find one another and share data. An immersive panoramic call begins with one of the participants requesting a call to the other via a matchmaking server — the server then is able to locate the accepting partner via their pre-registration with the server. The call then proceeds in two phases: an initialisation phase where partners set up communication channels and exchange metadata, followed by the per-frame loop where live environment data is sent.

3.2.1 Call Participants

Regardless of who initiates a call, we denote the participant using the mobile phone to be the *sender* and the HMD user to be the *receiver*. In literature of other shared-environments systems these participants may also be referred to as the *remote* and *local* user respectively [40] [60].

3.2.2 Server and Initialisation

During the main part of the call, wherein users are actively sharing their environment, communication between partners is directly peer-to-peer — that is, they communicate via internet infrastructure with one another without any further mediation. However when a call is placed the caller will not yet know where to send the call request. This is because the internet protocol (IP) address of the call receiver is often assigned dynamically, whether

by a wireless router or the mobile internet provider. For this reason users who wish to be able to participate in panoramic calls must notify a matchmaking server of their IP address as it changes; conceptually, the server stores a table of users to IPs so it can reply with a receiver address when another user requests a call to be placed.

In practice things are not so simple — we have decided to use the WebRTC library[¶] to alleviate some of the difficulty, but peer-to-peer communication remains an inherently complex process. Figure 3.5 shows the complete communication sequence required between sender, server, and receiver to initialise communication using WebRTC protocols. Due to a technique called Network Address Translation (NAT) [53], devices not directly connected to the internet – which will include nearly all wireless devices connected to a WiFi router or mesh network such as 3G – actually have two IP addresses, one each for the Local Area Network (LAN) and Wide Area Network (WAN). This creates two problems when trying to establish a peer-to-peer connection. First, devices can not reliably determine their own WAN address without aid from a third party. The STUN (Session Traversal Utilities for NAT) module [62] of the server helps to resolve this by accepting requests and replying with the requesting address and port, which will be the WAN address plus a port that allows reaching the devices through the NAT router. Secondly, even if we know both IP addresses of the partner to call, we cannot route a request to them without the previously mentioned NAT port, which is transient. Therefore as shown on the diagram, even though potential partners are preregistered they must each perform a STUN request during call initialisation.

As part of the WebRTC standard, this initial exchange would normally also include negotiation of device capabilities to allow for communication between devices with different input and output abilities. For our research we ignore this as we know our target devices are each capable of recording and displaying HD video. We skip straight to exchanging predetermined metadata regarding audio and video capabilities using a format called Session Description Protocol (SDP) [18], which when combined with IP address and port provided by STUN allows both sender and receiver to select an Interactive Connectivity Establishment (ICE) candidate [48] — the final agreement about how to communicate.

Finally, Figure 3.5 includes the TURN (Traversal Using Relays around NAT) module [37], which acts as a fall-back option in case the preceding process fails and a peer-to-peer connection cannot be established. The TURN module simply provides itself as the peer to each client, and proxies the video, audio, and data streams. In our testing scenario

[¶] `https://webrtc.org`

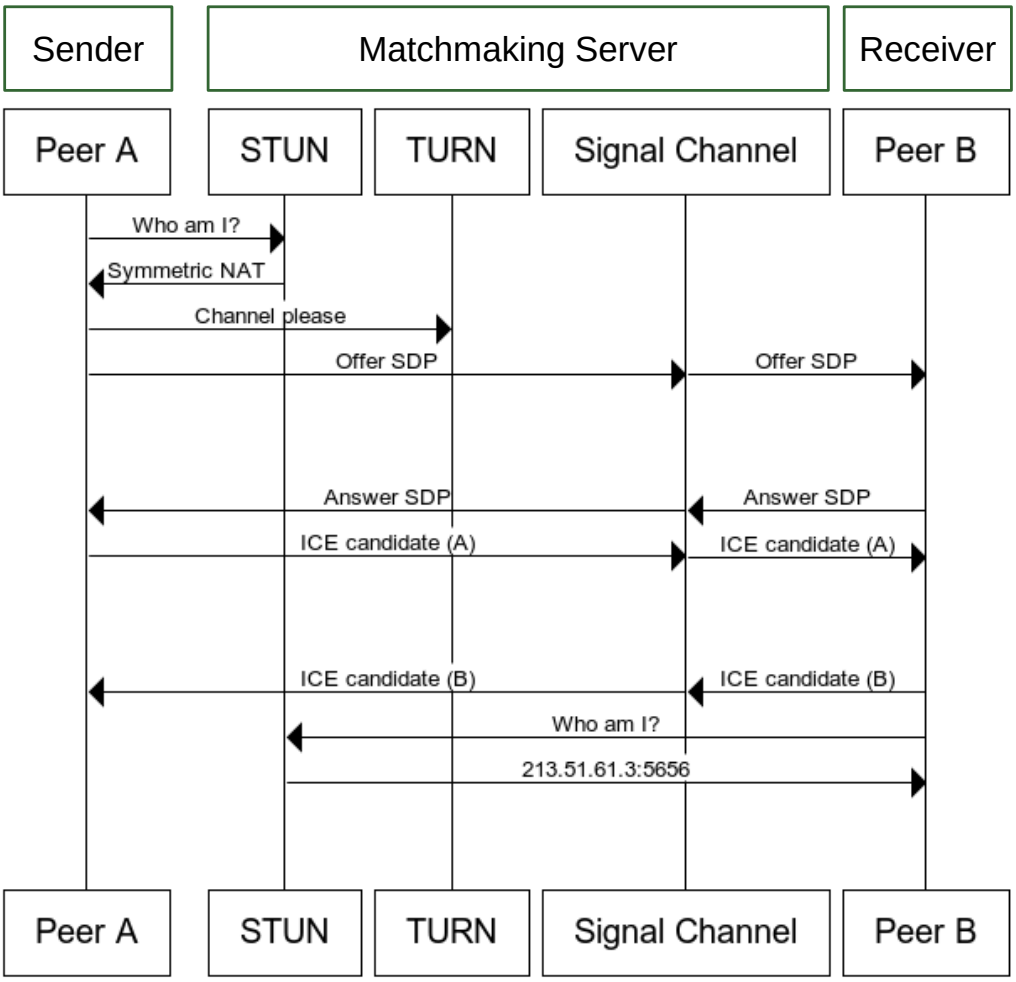

Figure 3.5: Sequence diagram showing the communication between Sender, Server, and Receiver during call initialisation. The green boxes (top row) show our call participants, the next row shows components of the WebRTC API. (adapted from [39], used under Creative Commons License)

STUN worked correctly, so we have disabled TURN. In a research context using the TURN module has little practical effect, just a slight increase in the end-to-end latency of the system. In a production system using TURN may require an extreme amount of bandwidth provision as all video streams are centralised to the provider's servers, however it would broaden the range of possible networks and devices the system could work on.

Once the WebRTC initialisation is complete, prior to opening the video and audio channels, we send a small amount of configuration data from sender to receiver during the call setup phase, namely the calibrated intrinsic camera parameters of the sender's camera.

3.2.3 Video Communication

Even though we do not directly show the environment receiver a live stream from the sender's camera, the frames are still conveyed in real-time over the network using a WebRTC video stream. Besides taking an additional copy of each video frame for computer vision prior to sending it, the 'send side' of the system operates much like any video calling application, i.e., a frame is captured from the camera, compressed using a video codec, timestamped, then sent to the calling partner using the UDP network protocol. For our system we use the H.264 video codec [61] and disable adaptive quality based on available bandwidth, locking the camera to 1280×720 (720p) to meet the quality target for our HMD user.

3.2.4 Audio Communication

As part of the immersive call experience we provide bi-directional audio communication using the phone's microphone and speaker. Again this uses a WebRTC stream established during call initialisation. Our system uses the Opus codec [59] to compress voice recording before transmission.

3.3 Representing the Environment

To allow the HMD user to control their viewpoint independently the environment must be stored in a spatially registered format. One option would be to use a 3D data format such as a point cloud or mesh data that is generated by the sender using a technique such as SLAM (simultaneous localization and mapping) [8]. This approach would favour smaller environments as it allows for rendering with stereo depth in HMDs, and due to

how SLAM works will tend to have fewer errors for closer features in the environment. Conversely, one could store the environment in a 2D map – a panorama – and project the environment to and from that map. This is the approach we have pursued.

A panorama is a representation of an environment that provides a wide-angled view projected down to a single two dimensional image. A spherical panorama extends this concept by allowing the environment to be captured from every possible angle about a fixed point, i.e., 360° horizontally and 90° both above and below the horizon. This can be contrasted to the more commonly used ([2] [4] [11] [42] [60]) cylindrical panorama, which only provides a limited view both above and below the horizon. We have decided to use a spherical panorama for our environments as unlike keyboard of mouse driven interfaces, the freedom of movement of an HMD does not prevent a user from looking directly up or down, so we want it to be possible to capture the environment in those directions.

Figure 3.6: Comparison of the visible area when viewing a cylindrical or spherical panorama. The entire map is available to spherical panoramas, whereas the darkened portions cannot be rendered with a cylindrical model.

By continuously generating and updating a spherical panorama on one device and exploring the real-time result on another we create a virtual remote location that can be viewed from any direction that incorporates live changes in a scene. The natural head movement of the HMD user controls the viewing direction inside the shared environment. In addition to the shared visual environment our system provides audio communication

in a similar fashion to existing mobile video call applications.

Figure 3.7 shows a panoramic environment represented both as 3D sphere and a 2D map. This representation is ideal for environments with predominantly distant features — primarily outdoors though large interiors such as a concert hall would also be suitable. This is because depth information is lost during panorama creation and the scene is stored in a 'flattened' format; when the environment is later viewed it will appear more natural if objects and features are outside the range of stereoscopic depth discrimination of around 18m [3].

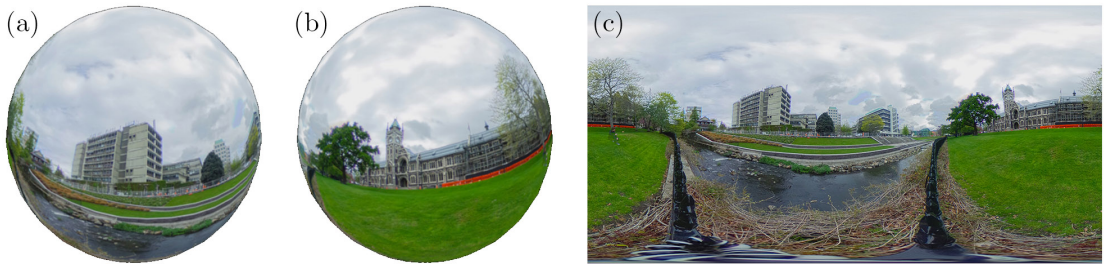

Figure 3.7: Spherical panoramas can be 'wrapped' to spheres as seen from the outside front (a) and back (b). The 'unwrapped' equirectangular projection is shown as (c).

There is also a practical consideration when using phone cameras to acquire images of the environment; we assume that the capturing camera position is fixed and the sensor rotates in-place around this fixed position. In reality the user will likely be holding the phone at arms length with rotation centred around their body, causing positional change of up to one arm span. The assumption of no movement is therefore only safe for distant scenery as the relative amount of camera movement compared to the distance of objects captured is too large for noticeable errors in the generated environment. As objects get closer and this distance ratio decreases there is increased potential for errors when estimating rotation from captured vision, and therefore errors in the panorama. For this reason when conceptualising the size of the spherical environment we treat it as infinitely large — all points on the sphere are as far away as possible. This makes the environment truly position insensitive, at the cost of not being a useful model for close objects in the environment.

3.3.1 Live Panoramas

The traditional approach of building a panorama is that once new vision of the scene is correctly registered into the 2D map space, only new visual information (the non-overlapping portion of the panorama and new frame) is written to the panorama. To create *live panoramas*, which reflect changes in the scene in real-time, we alter the generation method by including all captured vision in the panorama, overwriting any previously mapped portions of the environment. By doing this, motion in the scene that is captured can be observed by a viewer looking in the right direction at that time. Areas outside the capture region behave the same as a static panorama, simply showing the most recent view of the environment.

3.4 Frames of Reference

One of the goals of a shared environment is to invoke the feelings of spatial presence and copresence, meaning that the sender and receiver should feel as though they are in the same environment, together with each other, and the spatial model they each build of the environment should be the same. In our spherical environment the users are not simply close in the environment — they share the exact same, immovable position. Our goal then is to resolve the real-world scene, camera rotation, and HMD rotation in such a way that users can be looking in the same direction and seeing the same thing. We do this through a series of projections and linear transformations involving the following coordinate systems.

The first coordinate system to consider is that of the real world environment to be captured. It does not matter where in the world we are, so an absolute point of reference (e.g. as provided by the GPS) is not required. It is important however that the sky is projected to the top of the spherical panorama and that the horizon is central and flat; this will allow a natural viewpoint for the HMD user. To achieve this we use the measurement from the phone's gravity sensor and assign the world positive y direction contrary to the force of gravity. The x and z axes could then be placed arbitrarily to form a right-handed coordinate system, though we use the phone rotation at call initialisation to set positive z such that the camera is viewing along that axis. This has the additional benefit that a call receiver does not have to reorient themselves to see the initial view of the scene if we set their default forward axis to positive z as well. Calls will start with both viewers looking in the same direction, though not necessarily the same pitch to allow a natural horizon.

When the scene is captured with the camera the imagery is stored in a simple 2D coordinate system with the origin in the centre of the camera sensor and a width and height equal to the camera resolution. We then use the pinhole camera model and the calibrated camera intrinsics to project this image onto the inside of a unit sphere, with the centre of the projection looking down the positive z axis. This sphere is then transformed by the inverse of the camera pose (combination of rotation and translation, the latter of which we assume is 0) before being projected back to the panorama map. This panoramic map uses the equirectangular plate carrée projection, meaning there is a $1:1$ relationship between the x and y axes of the panorama and the longitude and latitude of the corresponding point on the sphere.

The panoramic map is the authoritative frame of reference for sharing the environment between sender and receiver, as the coordinate system remains constant with only the contents changing. As long as the mapped image is synchronised between call participants it is possible to share a consistent environment. On the receiver side, the panoramic map is projected once again onto a unit sphere with right-handed coordinates, where positive y is up and positive z maps to the centre of the panorama. This sphere is then rotated based on the HMD pose, and a virtual camera projects a segment of the environment onto the 2D phone display twice — once for each eye.

Finally we have the device centric coordinate systems to consider. The Android APIs define two coordinate spaces: a phone-relative (local) right-handed coordinate system with the positive x axis pointing to the right of the display and positive z pointing out of the phone screen, and an Earth-relative (world) right-handed coordinate system where positive x points east and positive z towards the sky. These also apply when using the phone inside an HMD. The local system is not used in our current system implementation, but the world coordinate space is useful in environment generation for rotation estimation and verification, and it is the only source of rotation information used when viewing from the HMD. When we use the sensor pose transformations we must first apply a change of basis matrix to transform them from the Android world space to our sphere-space.

3.5 Environment Generation

The key to generating an accurate representation of the environment is in the rotation step between capturing a view to the camera sensor and projecting that image to the panoramic map. For this we need two ingredients: the calibrated intrinsic parameters of the capturing camera, and the camera pose. As we have acquired the intrinsic camera

parameters ahead of time in a robust controlled process, and we assume no positional change in the pose, the primary problem to solve in both our system and our research is acquiring an accurate phone rotation synchronised to the point in time each camera frame is captured.

We noted above that through the gyroscopes and accelerometers equipped on modern phones we can get an absolute rotation of the phone, however this reading alone is not accurate enough to create a pleasing, well-stitched panorama, such as that shown in Figure 3.6. Instead we use computer vision techniques to detect matching features in sequential frames and apply a mathematical model to determine the relative rotation between each pair of successive frames.

If two camera images observe a planar scene from different angles, we can relate the images to one another through a homography matrix, which would allow mapping one image to the other in 2D space [56]. In our case, as we conceptualise the features in the scene as being infinitely far away, we can consider all imagery captured by the camera as coplanar (this model is why, as noted earlier, close features can cause errors). The homography can be estimated with a minimum of 4 matching feature pairs, though we attempt to find many more to increase the overall accuracy. The details of this approach are thoroughly examined in Chapter 4.

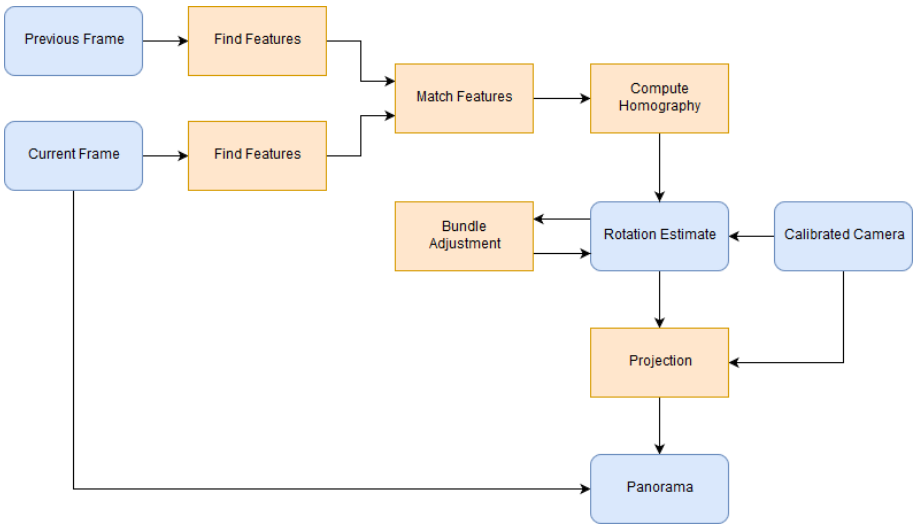

Figure 3.8: Overview of the process used to create the panorama representing the remote environment. The details of each step are explained in Chapter 4.

3.6 Environment Viewing

The receiving partner in an immersive panoramic call is able to view the shared environment in a natural way by wearing a GearVR and turning their head towards the parts of the scene they wish to view. We retrieve the head rotation measurement from the onboard hardware sensors inside the GearVR, which are much more accurate than those on the inserted phone. Given this rotation we can project the panorama contents using a virtual camera in place of the user's eye, simulating the view from inside the sphere.

The optical system inside the HMD is designed such that the user sees each half of the phone display with one eye, with the apparent focal distance a comfortable 2m away, rather than the few centimeters of distance in reality. Figure 3.9 shows what is rendered to the screen by our application — the optical system corrects the aspect ratio for a natural viewing experience.

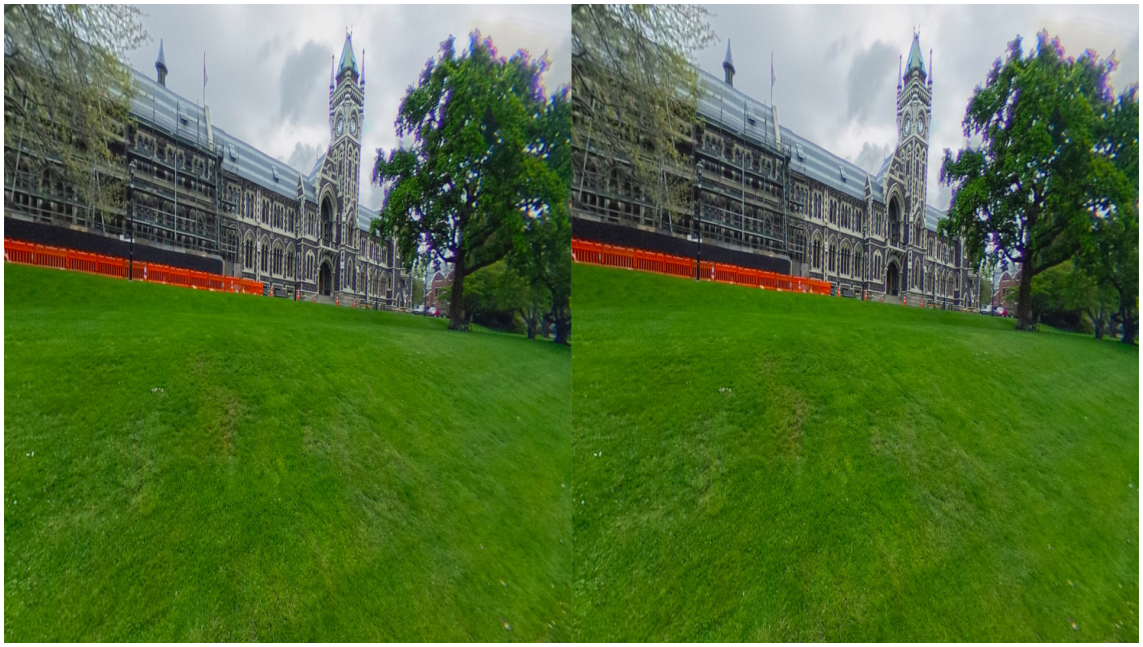

Figure 3.9: The final output as rendered on the mobile phone seated inside the GearVR. This rendering is monoscopic, meaning the same image is shown to each eye. The lenses inside the device correct the aspect ratio for the user.

Chapter 4

Implementation

Contents

In this chapter we provide the implementation details and specific mathematics that drive our immersive panoramic call system. We also describe the process of implementation to highlight some of the technical choices we made and their accompanying trade-offs, along with avenues which were explored or implemented but did not make it to the final evaluated prototype as they were either not fast or robust enough.

4.1 Development Requirements

In order to create our prototype system we require both development and testing hardware along with software to build and deploy the application. During runtime we also require a machine to act as a call matchmaking server and the attendant network infrastructure.

4.1.1 Hardware

During development we used two standard desktop workstations — one for programming work and one to act as the dedicated server. The latter machine was not connected to the internet but rather to a D-Link wireless router which provided a dedicated wireless network solely for our test devices. We also required phones to deploy our system to; whilst the complete performance evaluation was only performed on the LG Nexus 5 and Samsung Galaxy S7, we also tested calls on the Google Pixel, Huawei Honor 5X, and Lenovo Phab 2 Pro.

Figure 4.1: Left to right: the phones we used for both development and evaluation, the Nexus 5 and Galaxy S7, followed by phones that can run our application but were not fully evaluated, the Pixel, Phab 2 Pro, and Honor 5X.

4.1.2 Developer Environment

We developed our system using Android Studio on a desktop PC with Arch Linux installed. The primary alternative to Android Studio for developing Android apps is Eclipse, though we did not test it as the Android Developer Tools plugin for Eclipse has been deprecated, and one of our primary goals is for the system we develop to serve as a useful platform for future development. The only other software requirement is to have the dependencies listed below installed.

4.2 Dependencies

In this section we outline the dependencies required to build and run our applications. These libraries greatly reduce the time needed to develop our prototype by providing

functionality for common programming requirements. All the dependencies we have included are free and open source, meaning we have no restriction on their usage and can modify them as needed.

4.2.1 Android NDK

To gain precise control over the performance characteristics we developed the most computationally intensive components in C++. When programming for the Android platform using Android Studio the initial project setup only allows the use of the Java programming language. To enable C++, we must include the Android Native Development Kit (NDK), which is provided as a set of C++ header files and precompiled libraries. This in turn requires writing some bridging code using the Java Native Interface (JNI), which is C code where functions are given specially formed names that allow discovery by the Java runtime. Finally, to include the C and C++ code in the building and packaging steps we needed to replace the default build system `gradle` with `gradle-experimental`. As the name implies this tool is under active development and thus has missing features and can sometimes be unstable, but there is no alternative for writing C++ code for Android without using already deprecated tools.

4.2.2 WebRTC

The WebRTC (Web Real-Time Communication) library* provides functionality for transmitting video, audio, and binary data between two or more devices in real-time over the internet. It is intended as a 'full-stack' solution, that is, it uses a minimal interface to take care of all required functions end-to-end, including video and audio capture, transcoding, transmission, decoding, AV synchronisation, and display.

WebRTC is the most demanding of our dependencies — in fact its inclusion in the project has a flow on effect on the build process for the application and all other dependencies. This is because the WebRTC library source code is inseparable from its parent project, Chromium†, the open source base for the Google Chrome browser. To use the WebRTC library on Android we used a series of tools developed by Google: first `gclient` to download the 24GB repository of source code and dependencies, followed by the `GYP` tool to create a local build profile, and finally `ninja` to run the build.

One of the downloaded dependencies is Chromium's customised Android version of

*`https://webrtc.org`
†`https://www.chromium.org/Home`

libc++[‡], an implementation of the C++ standard library. As any given project should not link against more than one standard library, the inclusion of this dependency meant we had to recompile our other dependencies and the whole project against the customised library.

Once compiled, the WebRTC library consists of static libraries, shared libraries, C++ headers, and Java class packages. Because of this it was quite time consuming to fully integrate WebRTC into our system. This was an acceptable trade-off however, as the functionality it provides would be extremely time consuming to implement ourselves, and all competing options were either too immature or had unfavourable licensing requirements.

4.2.3 Camera2

We did not directly use WebRTC's full-stack API in our system, as the video capture module provided uses only the old Android camera API which does not allow for manual control of focus and exposure. Instead we replaced the video capturing portion of the stack with our own code that interfaced with the new `camera2` API. The WebRTC developers had intended for this to be straightforward in their design, unfortunately due to some errors in the library this was not the case at all. In fact, we had to resort to otherwise unfavourable software engineering practices to complete our implementation, such as using runtime reflection to overwrite private member variables, and reproducing undocumented memory structures to allow for modification of internal WebRTC state.

Overall, creating camera2-to-WebRTC interoperability was a significant piece of work but also a useful contribution to our goal of this system being used for future research. Challenges we faced included missing or inaccurate documentation, bugs in camera drivers to work around, and hardcoded references to the old Android camera API. Interestingly, we found comments in the WebRTC source code suggesting that a `camera2` module was desired but presently too difficult to bother with. At the conclusion of this thesis project we intend to refine our implementation and submit a patch to Google to include our work.

4.2.4 OpenCV

OpenCV[§] (Open Computer Vision) is an open source library that provides a vast amount of functionality under the broad umbrella of computer vision. For our purposes we are most interested in algorithms that provide feature detection, motion estimation, and camera

[‡] `https://libcxx.llvm.org/`
[§] `http://opencv.org`

projections.

OpenCV is a cross-platform library with pre-built static libraries for many desktop and mobile platforms. However, due to WebRTC's dependence on libc++ noted above we needed to compile our own version of the library to correctly link everything in the project. The OpenCV documentation claimed to support this configuration but this was not the case; we fixed this issue directly in the OpenCV codebase and submitted a patch to the maintainers[¶], which has since been integrated into OpenCV starting with version 3.2. This involved rewriting the build files for the target configuration and altering some system calls to use up to date standards.

4.2.5 Eigen

Eigen[‖] is an open source, C++ header-only library that provides data types and algorithms for performing linear algebra. It can be used simply be downloading and then including the provided header files — no initial build step or project configuration is required. We use Eigen for all matrix maths that is not handled by OpenCV.

4.2.6 OpenGL

Whilst not its own independent dependency, it is worth noting much of our implementation is performed on the graphics processing unit (GPU) through vertex and fragment shaders, which in turn requires a graphics API. We used the `android.opengl.GLES20` package provided by Android Studio as part of the base SDK, which provides OpenGL 2.0 GPU functionality. This release is a major version behind the most up to date version (3.0), but it is ubiquitous on Android devices and provides all the functionality we need. In particular it allows us to access the programmable graphics pipeline, allowing us fine control over the rasterisation process and to specify custom vertex and fragment shader input formats.

4.3 System Architecture

As described in Section 3.2, during a call we use a peer-to-peer network architecture, though a server is required to be running to locate peers and initialise a call. Additionally, for ease of prototyping we created separate apps for each of the sender and receiver — in a more complete deployment these would be merged into a single app and take on the

[¶] `https://github.com/opencv/opencv/pull/7174`
[‖] `http://eigen.tuxfamily.org`

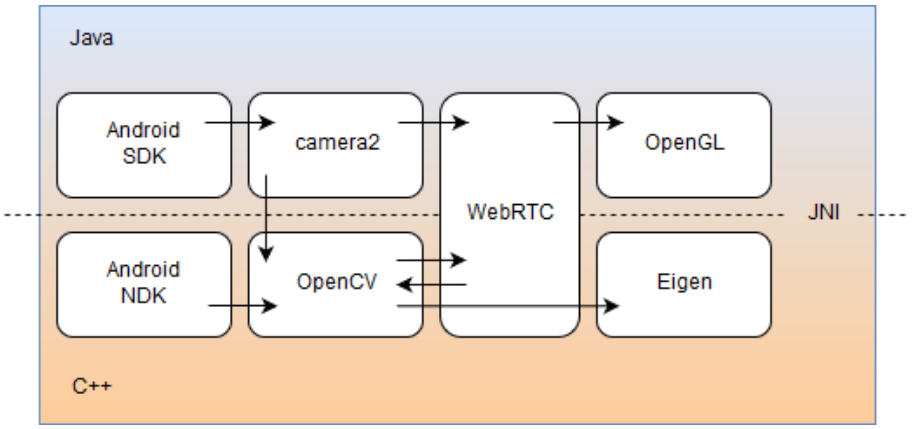

Figure 4.2: Overview of required dependencies with their relationship to the Java/C++ JNI boundary. Arrows show direct data flows between APIs of each dependency.

required role based on whether the user intended to capture their environment or receive a shared environment.

4.3.1 Client Software

We deployed our system as two separate Android apps which shared only networking code between them. We created an app called 'SphereMapper' for the remote, phone-only user who plays the sending role in a call, and 'SphereViewer' for the receiving HMD user. Each app has the following operation method: when the app is launched, it notifies the server it is ready to make or receive a call. Once both parties are ready, the software negotiates with the server to establish a peer-to-peer connection and the call begins. For the HMD user there is no interface besides using head movements to look around inside the shared environment as it becomes available. The sending user can use a finger swipe to choose between a full view of the scene as captured by the camera or the equirectangular representation of the shared panorama, which can be used as a guide for where new parts of the scene should be captured.

4.3.2 Server Software

The server software is an application written in the Go programming language which conforms to a reduced version of the WebRTC specification. In particular, it provides a STUN service as described in Section 3.2.2, and provides message forwarding to help establish a peer-to-peer video connection. It operates on a first-in first-out matching basis,

Figure 4.3: The sender user interface showing the view of the camera fullscreen, along with a manual exposure slider.

that is, when a SphereViewer client connects they will be connected to the first pending SphereMapper connection in the queue, and vice versa.

4.4 Establishing a Call

At application launch each of the sender and receiver sends a key-value message to the server notifying it of which role they will be taking in a call. If the server has seen a user with the opposite role that has not yet matched a partner it will respond with a message notifying the new connector that session negotiation can begin. A peer-to-peer session is negotiated using the Session Description Protocol (SDP) [18], with initial messages proxied through the server and a final direct message confirming the connection. Refer to Section 3.2.2 for more details.

With peer communication successfully established we use WebRTC APIs to open three separate data streams (called channels in WebRTC), one for each of video, audio, and data. We use the data channel to immediately send the receiver some information required

Figure 4.4: The sender user interface in panorama mode, showing which parts of the map have been built. The button in the lower right exports the panorama and metadata for testing.

during the environment sharing process, in particular the intrinsic camera properties of the sender. Once the video channel is open for transmission, we launch our custom `camera2` module to begin capturing the environment, and being the panorama generation and sending procedure described below. We attempt to guess the correct exposure level from the initial frame of the scene, however the camera view interface also includes a slider that allows changing the exposure at runtime.

4.5 Generating and Sending Environment

When a call is established the participant building and sending the environment determines an estimate for the phone rotation at the time of each video frame capture and sends this pose data concurrently with the video frame pixel buffer. This estimate-and-send procedure is executed in a loop that attempts to keep pace with the camera update rate — 30 frames per second on our devices. If the loop is unable to process the video stream

at this rate any frame queued for processing will be discarded as a new frame becomes available.

4.5.1 Acquiring Camera Frames

As previously noted, we decoupled WebRTC's internal camera acquisition code in order to use our own `camera2` module to enable fine control of the camera driver, in particular the settings for auto-exposure, autofocus, and white balance. This was not a straightforward as expected — the API is designed for the specific use case of displaying the the camera feed immediately to the user's screen, whereas we want direct access to the pixel buffer from C++. This is further complicated by our desire not to incur performance penalties from copying data, but rather having direct pixel buffer access.

We built a set of C++ and Java classes that ended up getting direct memory access to the camera sensor output data, though this required some engineering complexity including thread synchronisation, inspecting Java object memory layout from C++, and working around some camera driver issues (which included returning data in the wrong format, locking access if queued frames weren't acknowledged in a timely manner, etc).

4.5.2 Pose Estimation

We considered two possible categorically distinct methods for determining the pose of the sending phone — internal sensors and computer vision. Instead of directly querying the internal sensors we used a fusion of multiple sensors and Kalman filtering as described in [43] to acquire a sensor pose. Whilst we found that sensors alone are insufficient for getting an accurate enough pose to build a seamless panoramic environment, the data they provide can be used to enhance computer vision results. The remainder of this section describes our computer vision approach, which is the major contribution presented in this thesis. Much of the fundamentals are based on well known processes as summarised by Szeliski [56], which we have modified both due to our application use case and to fit within the performance constraints of mobile phones.

For the remainder of this section frames and pixel buffers refer only to luminosity values in those buffers — when performing computer vision operations we completely disregard any colour information from the camera output. Because Android cameras are able to provide YUV formatted buffers in separate planes (i.e., not interleaved in memory) we can simply discard the U and V data to obtain a luminosity map.

On the first frame we set the absolute pose $\mathbf{P}$ based on the initial estimate $\mathbf{P}_E$ from

the sensor fusion implementation. We decompose the sensor estimate into three rotational components pitch, roll, and yaw, then create our first pose $\mathbf{P}_1$ from just the pitch and roll as follows

$$\mathbf{P}_E = \mathbf{R}_z(\alpha)\mathbf{R}_y(\beta)\mathbf{R}_x(\gamma) \tag{4.1}$$

$$\mathbf{P}_1 = \mathbf{R}_y(\beta)\mathbf{R}_x(\gamma) \tag{4.2}$$

where

$$\mathbf{R}_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$$

$$\mathbf{R}_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The reason we use this partial rotation is that we want the sky to be up and the horizon to be level (as explained in Section 3.4) – which can be determined with reasonable accuracy by the sensors – but we also want the initial captured vision to be in the centre of the panorama as that is where the receiver will be looking when the call starts. This prevents them having to look around inside the spherical environment to locate where the environment is starting to be generated.

We also take a reference to the initial frame pixel buffer $\mathbf{p}$ to be used when the next frame becomes available. As new frames become available we then begin the main computer vision loop. Our aim is for each pair of subsequent frames to be processed in under 33ms, to align with our initial requirement of a frame rate of 30 FPS. Note that this needs to be a constant frame rate – not average – as we need each frame estimation to be accurate before sending to the receiving user, so there is limited opportunity for offloading work to alternate frames or to employ a multithreaded design for the computer vision component.

For each frame of the camera stream beyond the first we aim to use computer vision techniques to derive a rotation transform $\mathbf{R}$ that will map the preceding frame $\mathbf{p}$ onto the current frame $\mathbf{c}$ when each are projected to sphere-space. The first step we take is to downsize each of $\mathbf{p}$ and $\mathbf{c}$ by 75% of the original to 960×540. This is done using a separate,

preallocated memory buffer so the full frame is still available to send via WebRTC.

Using the downsized buffers $\mathbf{p}$ and $\mathbf{c}$ we find *features* in each image and store them as sets $\mathbf{f}_p$ and $\mathbf{f}_c$ respectively. A feature is pair of data that describes a location in an image (called the keypoint) and the immediate vicinity of the keypoint in terms of intensity, contrast, size, and gradient direction (collectively, a descriptor). There are numerous keypoint detectors and feature descriptors available in both the research literature and through OpenCV's API; we used the ORB technique [50] through the `OrbFeaturesFinder` class. This method internally uses FAST [49] keypoint detection and BRIEF [9] descriptors with parameters set to optimize for minimal CPU usage, which fits our use-case well. Another benefit of using FAST features is that unlike alternatives such as SIFT [35] and SURF [5], the FAST algorithm is unencumbered by patents, making for less concern for future use as the research platform evolves.

FAST keypoints are found based on their suitability for tracking, that is, we should be able to find the same image features in consecutive frames even as they change position in the camera sensor due to phone rotation. In general this means points that have nearby areas of high contrast. The BRIEF descriptors are used to tell features apart without resorting to expensive pixel-wise comparisons. If two keypoints from distinct frames have a descriptor similarity above a certain threshold we can say that they are most likely the same point on a real world object. The modified BRIEF descriptors used by ORB also have the property of rotation invariance — we can still classify two features as the same even if they are observed from a different angle, which is a likely scenario when creating a panorama.

For our implementation we modified the ORB detection process in two ways to increase performance. First we modified the kernel size parameter of the FAST detector from the default of $9:16$ down to $7:12$, which on average finds 85% of the keypoints in 80% of the time. We also modified some memory management aspects of the BRIEF algorithm — part of the process involves creating an image pyramid, which is a series of downsampled images of decreasing size based on the original large image. We changed the OpenCV implementation to store the pyramid images in memory we preallocate at application startup, thus avoiding expensive memory allocation operations each frame.

With the sets of features $\mathbf{f}_p$ and $\mathbf{f}_c$ now available, we try to match as many features from $\mathbf{f}_p$ to their corresponding partner in $\mathbf{f}_c$ as possible. We use a simple brute force matching strategy, that is, we compare every feature descriptor from $\mathbf{f}_p$ with every entry in $\mathbf{f}_c$ and assign a similarity score. We keep the pair with the highest similarity for each feature and

add it to the set of matches **m** provided it also satisfies the following two criteria: the similarity score is above 70% of the highest possible, and the score is at least double the next best matching pair. The latter condition helps to filter out matches where a feature in **p** could potentially be matched to multiple locations in **c** — common in scenes with repeating patterns and textures.

We now use our set of matches to derive a homography matrix **H** — refer to Section 3.5 for the principals of homographies or [56] for a more complete treatment. We simply use the `HomographyBasedEstimator` class provided by OpenCV for this step; the underlying implementation uses four matching pairs from **m** to estimate a homography, then uses the RANSAC [14] process to iteratively improve the estimate. This involves deriving an estimate from random pairs of matches and comparing the estimate against the remainder of the set **m** and counting the inliers. New random input pairs are chosen until the estimate can no longer be substantially improved.

By initialising the `HomographyBasedEstimator` to use a model that conforms to pure rotation (disregarding translation) we can now use the camera matrix **K**, derived from the camera calibration described in Section 3.1.2, and the pinhole camera model to create an initial rotation estimate $\mathbf{R}_i$ per [19]

$$\mathbf{K} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

$$\text{where} f = \text{focal length}, p = \text{principal point}$$

$$\mathbf{R}_i = \mathbf{K}^{-1}\mathbf{H}\mathbf{K} \tag{4.4}$$

The last step in our computer vision pipeline is to refine the rotation $\mathbf{R}_i$ using a technique called bundle adjustment. Again we take advantage of OpenCV's implementation with the `BundleAdjusterReproj` class, which refines $\mathbf{R}_i$ with the aim to minimise the sum of all reprojection error caused by mapping each element of $\mathbf{f}_p$ to $\mathbf{f}_c$ using $\mathbf{R}_i$. We take the final result of this process as the rotation between the two frames **p** and **c**, **R**. We can now simply update the absolute pose **P** by combining the existing pose from the previous frame and this new relative rotation

$$\mathbf{P}_{n+1} = \mathbf{R}\mathbf{P}_n \tag{4.5}$$

Before committing to this pose as the correct result for the current frame we perform

a final check using the relative rotation estimate $\mathbf{R}_E$ from our sensor fusion module. We can find the difference between our computer vision rotation and the sensor estimate by multiplying one rotation matrix by the inverse of the other, yielding a new rotation matrix $\mathbf{D}$ that describes the difference between the two. Note also that because rotations are orthogonal matrices $\mathbf{R}^T = \mathbf{R}^{-1}$. Furthermore, we can quickly compute the magnitude of the difference by examining the trace (sum of diagonal elements) of $\mathbf{D}$, as in

$$\mathbf{D} = \mathbf{R}^T \mathbf{R}_E \tag{4.6}$$

$$\mathrm{tr}(\mathbf{D}) = 1 + 2\cos\theta \tag{4.7}$$

Taking these together with a threshold of 5 degrees, we can discard both the frame and estimate where

$$\mathrm{tr}(\mathbf{R}^T \mathbf{R}_E) < 2.99239 \tag{4.8}$$

It is quite important to drop these potentially erroneous frames as we are only using relative tracking from the second frame onwards — there is no easy way to recover from a substantial rotation misestimate, all future frames will be derived from it, so it is safer to simply discard it.

At this stage we have both the camera pose and a frame to send to the receiving user — we copy the pose data to the WebRTC thread (which already has a copy of the full colour frame) and send it over the network as described in the section below. Finally, before iterating this loop again we drop the reference to the frame $\mathbf{p}$ and assign $\mathbf{c} \to \mathbf{p}$, and likewise update the current set of known features in that frame $\mathbf{f}_c \to \mathbf{f}_p$. For future frames beyond the second we can therefore avoid both the downsampling and feature detection and description steps for $\mathbf{p}$. This step is omitted if the frame is dropped based on the sensor estimate, meaning the new incoming frame will be compared against the frame two steps behind instead of the usual one.

4.5.3 Sending Data

Once we have received a frame of video and computed the absolute pose of the camera at the time of capture it would be ideal if we could send these to the call partner combined into a single message stream. Unfortunately, WebRTC requires that the video stream carry only VP8 or H.264 encoded video data (in our case the latter), so we must pass the pose information by a separate data stream. Whilst WebRTC provides a method for precisely synchronising video and audio streams, no such facility is implemented for video

and data. To complicate matters further, there is no method of uniquely identifying a given frame between a sender and receiver — WebRTC strips all frame level metadata such as sequence number and timestamp immediately prior to sending frames via the network. For this reason we developed a visual tagging system to synchronise pose information with the video frame it belongs to.

We assign each frame a tag between 1 and 255 inclusive, incrementing each frame and wrapping back to 1 on overflow. We avoid using 0 as a possible tag as malformed frames can be decoded to a buffer of 0 data, which we do not want to translate into a valid tag. To transmit the tag we replace a section of the Y (luminosity) channel data prior to video encoding to embed the tag value — for this reason it must be both temporally and spatially resistant to modification during the video encoding process. To avoid spatial disturbance we write the tag data to two 8×8 squares in the top left of the image, expanding 1 byte of tag data to 128 bytes of Y data being replaced. To avoid temporal compression causing incorrect tags, we do not simply write the tag value t directly in the Y channel but instead write the high 4 bits to the left square and the low 4 bits to right, each multiplied by 16

$$T_{\text{left}} = 16 \left\lfloor \frac{t}{16} \right\rfloor \tag{4.9}$$

$$T_{\text{right}} = 16(t \bmod 16) \tag{4.10}$$

This allows for a significant luminosity difference between frames despite the tags being consecutive integers. In practice this has shown to be a highly robust method for identifying frames.

To avoid the visual artefacts caused by adding the tag to the video data, prior to overwriting we extract the existing Y channel data for the affected 16×8 region. We can then add this data to the message for the frame, allowing the receiver to rebuild the frame to match the original state.

With the tag, pose, and extracted Y data now all available we compose a MessagePack** message of the combined data and send it on the data stream at the same time as we send the tagged video frame to the video stream.

**`http://msgpack.org/`

4.6 Receiving and Viewing Environment

During a call the viewing participant will receive a continuous stream of tagged video frames and messages describing the pose of each frame. Due to the relative size of video frames compared to data messages, we observe in practice that the data will always arrive first. For that reason when new data arrives we take no action except to store the message in a map of tag → data. If there exists some data with that tag already stored we simply discard it as it will be at least 8 seconds old (from frame time × unique tag amount).

4.6.1 Synchronising Video and Data

When a video frame is received we first determine its tag. Because of the jitter introduced by video encoding, to add robustness we examine each Y channel value in the tag region, round to the nearest 16, then count the most frequently occurring rounded value. When divided by 16 this value will give us the tag high and low bits from the left and right regions respectively, which can then be assembled into the full byte sized tag.

If a corresponding entry is not stored in the tag map, we assume that either this is a malformed frame, a duplicate frame, or has arrived especially early or late. In all these cases the remedy is to simply drop the frame and wait for new incoming frames. For two phones connected via a common WiFi network we have empirically determined these occurrences are extremely rare — less than 1 frame per minute is dropped on average.

In the common case of a successful tag match, we replace the Y data in the tagged region of the video frame with the Y data sent with the tag message, undoing the data loss incurred by the tagging process. We then convert the YUV data to RGB colour with a fragment shader writing directly to a framebuffer in video memory using the following equations

$$R = Y + 1.403(V - 0.5) \tag{4.11}$$

$$G = Y - 0.344(U - 0.5) - 0.714(V - 0.5) \tag{4.12}$$

$$B = Y + 1.77(U - 0.5) \tag{4.13}$$

then we hand the resulting GPU buffer to a separate rendering thread along with the corresponding pose information. The rendering thread will then update the stored panorama as described below.

4.6.2 Generating and Updating the Live Panorama

Unlike a static panorama, when updating the live panoramic environment we completely replace the existing panorama contents underneath the new region covered by the camera. We also assume that that the pose provided has been completely refined; we do not read anything from the existing panorama buffer for the purposes of stitching or alignment.

The panorama itself is stored in GPU memory as a series of texture-backed framebuffers. Splitting the framebuffer is a technical requirement enforced by the OpenGL implementation — we cannot create textures with a side larger than the value of `GL_MAX_TEXTURE_SIZE`, which for both our phones is 4096 pixels. To choose the desired size of the panorama we used the average horizontal field of view of our phone cameras (40°) to see how many full frames would be required to fill the panorama, then multiplied by the frame width, i.e.

$$1280\frac{360}{40} = 11520 \tag{4.14}$$

We then need to round this value to a power of 2 to fulfill OpenGL's framebuffer requirements, with the candidates being 8192 (2^{13}) or 16384 (2^{14}). We chose the former as the performance loss was not worth the marginal increase in resolution offered by the higher resolution. Finally, the height is determined by our use of equirectangular projection and must be half the width, giving a total panorama resolution of 8192×4096.

The first step in the mapping process is to quickly narrow the potential area that will be written to. Because the core of the panorama rendering is implemented as a fragment shader, each pixel must individually run through a non-trivial amount of processing before we can be certain whether it will be drawn. Due to the large size of the panorama buffer we would not be able to process incoming frames fast enough using this direct approach. Instead we overlay the panorama with a grid of rectangles (culling quads) and precompute the position in sphere space that the corners of the rectangles represent when projected from the panorama. For a given camera diagonal field of view F we can determine if a rectangle will be fully or partially inside the camera's frame if for any of its sphere space corners $\mathbf{c}$

$$\mathbf{P}\boldsymbol{z} \cdot \mathbf{c} \geq \cos(\frac{F}{2}) \tag{4.15}$$

where $\boldsymbol{z}$ is the unit vector along the z axis. For those rectangles that have any corner which passes this test we proceed with the following algorithm.

Now that we have determined the approximate subset of texture area that needs to

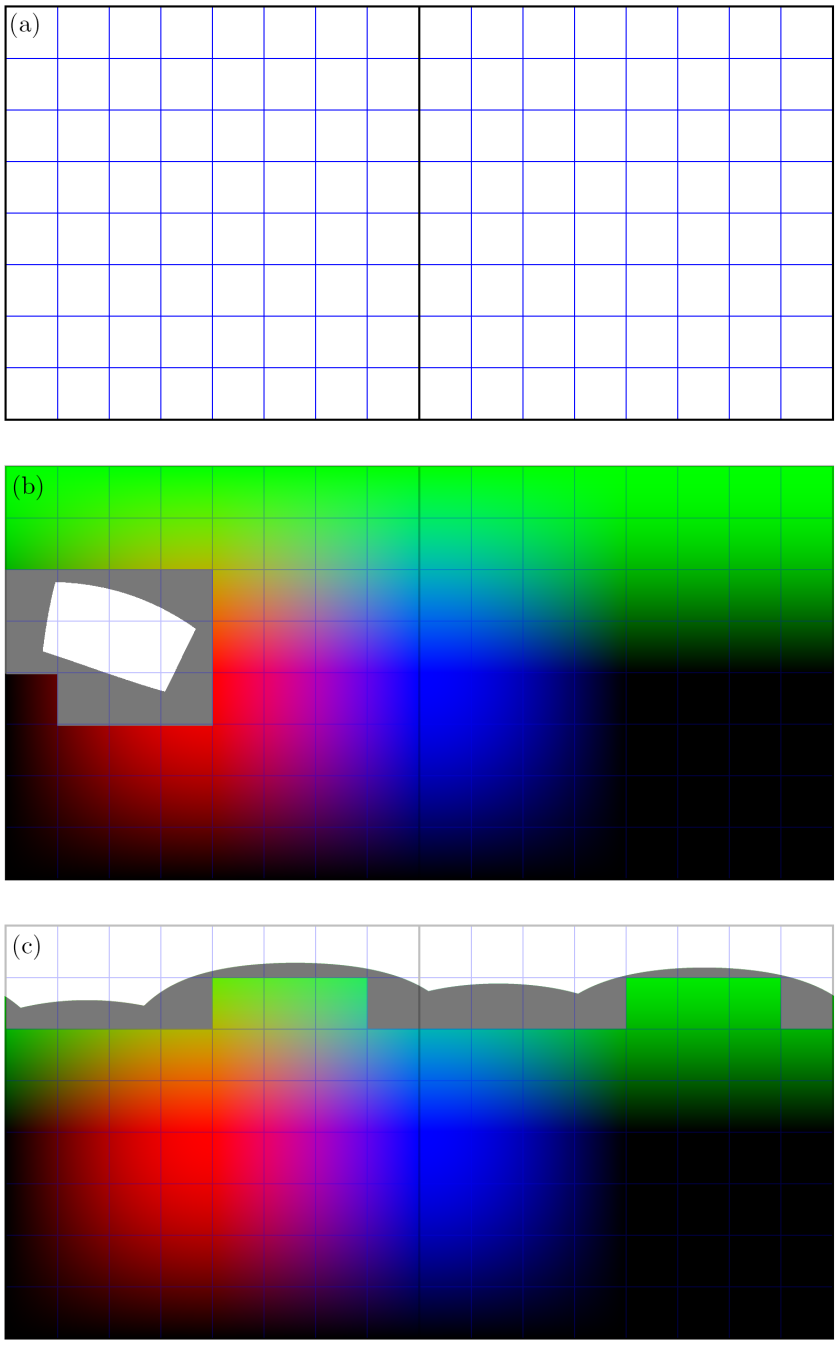

Figure 4.5: (a) shows the relationship between framebuffers and culling quads — the solid thick black lines are the borders of the framebuffers holding the panorama data, and the lighter blue grid shows the position of culling quads. (b) and (c) each show the relationship between $\mathbf{t}$ and $\mathbf{M}$, where red, green, and blue show the positive x, y, and z vectors respectively. Overlaid on (b) and (c) are white regions showing the camera's current view into the panorama, and the quads that run full fragment shader evaluation in grey. Note that looking at the poles causes over twice as much work as near the horizon.

be updated, we use a fragment shader to iterate over each pixel in that buffer area and potentially paint them with a the colour sampled from the incoming frame. The first step is to transform the panorama texture coordinates $\mathbf{t}$ to the equirectangular sphere map coordinates

$$\theta = 2\pi(\mathbf{t}_x - 0.5) \tag{4.16}$$

$$\phi = \pi\mathbf{t}_y \tag{4.17}$$

followed by projecting these to a unit vector on the environment sphere $\mathbf{M}$

$$\mathbf{M} = \begin{bmatrix} \sin\phi\sin\theta \\ \cos\phi \\ \sin\phi\cos\theta \end{bmatrix} \tag{4.18}$$

The next step is to rotate the sphere-space projection based on the incoming frame's associated pose $\mathbf{P}$, followed by a projection to the camera's sensor space using its intrinsic matrix $\mathbf{K}$, resulting in the camera coordinate for this pixel as

$$m' = \mathbf{K}\mathbf{P}^T\mathbf{M} \tag{4.19}$$

We now perform a secondary culling check, in this case because near the sphere's poles a pixel may be in both the camera's forward and back projection (appearing on the opposite side of the sphere), and we want to avoid rendering the latter. We do not proceed rendering the pixel if $m'_z \leq 0$.

As m' is in a homogeneous coordinate space, and is also in the camera's pixel coordinates, whereas we want a texture position on the range $[0, 1)$, we must normalise across both those factors. This can be achieved in a single step with

$$m = \frac{m'}{rm'_z} \tag{4.20}$$

where r is the 2D resolution of the camera sensor, in our case always 1280×720. This is the final result of where to sample in the incoming frame texture for the colour to apply at the source location $\mathbf{t}$ in the panorama.

If both the x and y coordinates of m are inside the range $[0, 1)$ we sample the video frame at that location and write the colour to the panorama coordinate $\mathbf{t}$. If either are outside the range, we simply take no further action. Unfortunately there is a performance cost to each pixel needlessly evaluated to this point, but due to the non-linear relationship

between the panorama coordinates and the spherical environment it is not possible to eliminate them all ahead of time.

Figure 4.6 shows a panoramic map with several frames rendered using this process.

Figure 4.6: A half complete panorama stitched using frame-to-frame tracking on a Galaxy S7.

4.6.3 Viewing the Spherical Environment

With environment capture, pose computation, network communication, and panorama generation all taken care of, all that remains is to show the HMD user vision of the remote scene based on the direction they are looking. We can think of the user's head in terms of a virtual camera inside a sphere that is textured with the panorama — though because we are just transforming one 2D input (the panorama) to a 2D output (the screen) we implement the environment viewer using just a quad and a fragment shader, rather than any rasterised spherical geometry.

The GearVR is designed as a stereoscopic device, that is, it can show a different view to each eye, which is achieved simply by splitting the mobile display directly down the middle as shown in Figure 3.9. Due to our intended use-case of sharing scenes with features outside the distance of stereoscopy, we instead use a monoscopic approach; we render the scene once to a framebuffer then fill the left and right halves of the screen with that same image. This has the welcome side effect of having a much simpler and more performant

implementation.

We aim to render the view of the environment at a constant 60 FPS regardless of how fast the sender is providing new frames from their camera for two reasons: first, the HMD user should be able to look around at the existing constructed environment regardless of whether it has changed, and secondly because rendering any slower will create a laggy, disjointed feeling for the user, which is likely to effect their sense of presence. Because of this we render the view in a separate thread to the one updating the panorama. Each frame begins with a query to the HMD for its current pose $\mathbf{P}$. Unlike using the phone sensors, the HMD inertial measurement unit (IMU) is of sufficient accuracy for estimating rotation due to more advanced hardware [29] and APIs that provides features such as future motion estimation.

Rendering the HMD view is essentially the same procedure as updating the panoramic map in reverse — we create a virtual camera matrix $\mathbf{K}$ to simulate an eye with 82° field of view and use a fragment shader to iterate over each output pixel as a unit vector cast from the centre of the sphere $\mathbf{M}$ such that

$$m' = \mathbf{K}^T\mathbf{P}^T\mathbf{M} \tag{4.21}$$

and once again normalise the result

$$m = \frac{m'}{m'_z} \tag{4.22}$$

to reveal where in the panorama to sample for colour to show at the given position in the output. This result is rendered initially to a framebuffer half the size of the screen so we can quickly render the final separate eye output by drawing a repeated textured quad.

4.7 Optimisation

All throughout our implementation we look for and take opportunities to reduce CPU workload as it is the primary constraint that can prevent real-time environment generation and viewing. Besides carefully crafting our implementation for efficiency throughout, we used the following specific methods to improve performance.

One of the key techniques employed is avoiding memory allocations. On both the sending and receiving applications there are loops which handle large amounts of memory in the form of image buffers — we take care to ensure the memory required is allocated before the loop begins to avoid allocations each frame. In some instances this involved

modifying the OpenCV source code to change the implementation of certain algorithms as buffers were allocated internally. We also preallocated C++ `vector` types with predetermined sizes, for example when storing feature descriptors or sets of keypoint matches. This has the side-effect of enforcing a maximum number of matches in those sets, which in turn limits the computational requirements, though at the risk of affecting accuracy.

We also take the opportunity to perform maths using the GPU instead of the CPU where possible. It is especially useful when processing large amounts of data in a loop where any given iteration does not rely on results from the previous, i.e., it is safe to parallelise the workload. As noted above, large portions of the algorithms employed are performed on either fragment or vertex shaders. An additional optimisation available when using the GPU is to take advantage or hardware linear interpolation; we do this for equations featuring only linear relationships by moving them to the vertex shader stage. For example, equations (4.16) and (4.17) each have a linear input and output, so when computed on a vertex shader are only processed at the corner points of the culling quads, whereas the following equation (4.18) is non-linear and must be computed for every output pixel in a fragment shader.

Finally, we utilise hardware threads for concurrent processing where possible. Whilst we can never keep all cores active at once due to the interrelation of data (the computer vision thread cannot proceed while waiting for new data from the camera thread), there were small performance boosts to be gained by having dedicated threads for each of computer vision, UI, shaders, networking, and camera frame acquisition. This is in addition to the threads created by WebRTC for audio and video encoding and network monitoring.

4.8 Alternative Approaches

During development of the final prototype described above other computer vision based approaches were implemented and briefly tested before being deemed infeasible due to either computational performance or mapping accuracy reasons. The techniques described in this section were not only considered, but partially or fully implemented, before being removed due to lack of performance or accuracy. Nevertheless, it is instructive to see what we tried that did not work out.

4.8.1 The Downward Spiral

The following techniques were not retained for qualitative evaluation because they suffer from what we term *the downward spiral*. This occurs when the time taken for the algorithm to return a rotation estimate is in part determined by how large the rotation is, and due to this the rotation of subsequent frames becomes larger because we acquire new frames from the camera with decreasing frequency. This feedback loop either leads to frames taking over 1 second to process (and in turn, Android drops access to the camera for not polling it often enough) or the estimate accuracy becomes no better than using the internal sensors.

4.8.2 Undistorted Frames

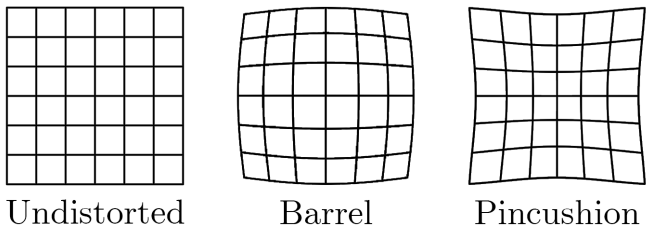

Figure 4.7: Diagram showing possible effects of camera lens distortion. Through the process of undistortion we return the image to the original state.

One feature we tried to add to our eventual implementation was to undistort camera images. Due to the lens design of the mobile phone cameras the actual image captured does not perfectly conform to the pinhole camera model we have used above, but rather will be affected by either pincushion or barrel distortion as shown in Figure 4.7. This distortion can be quantified by camera distortion coefficients k_1, k_2, k_3, p_1, and p_2, which are found through the same calibration process we use in Section 3.1.2.

Our first approach was to simply use the OpenCV APIs `initUndistortRectifyMap` at initialisation and `remap` every frame, which modifies the frame data in place to produce an undistorted image. This turned out to be much too slow, consuming up to 15ms on the Nexus 5 (almost half the time budget). A second option was to instead apply the undistortion mapping to only the found keypoints after running the FAST keypoint detector using `undistortPoints`. Once again this was unacceptably slow at 10ms.

A final approach was to implement the undistort algorithm in the same fragment shader that writes to the panorama as follows: given our camera parameters of focal length f, principal point c, camera resolution s, and the distortion coefficients above, we can find

the correct camera pixel location in normalised texture coordinates τ from the input pixel coordinate m with

$$\tau'' = (\frac{m_x - c_x}{f_x}, \frac{m_y - c_y}{f_y}) \tag{4.23}$$

$$r = |\tau''| \tag{4.24}$$

$$\tau'_x = \tau''_x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1\tau''_x\tau''_y + p_2(r^2 + 2\tau''^2_x) \tag{4.25}$$

$$\tau'_y = \tau''_y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2\tau''^2_y) + 2p_2\tau''_x\tau''_y \tag{4.26}$$

$$\tau = (\frac{\tau'_x f_x + c_x}{s_x}, \frac{\tau'_y f_y + c_y}{s_y}) \tag{4.27}$$

This provided a much more acceptable performance impact of under 2ms, but by implementing at this stage we now had a mismatch between the how the rotation was estimated (in distorted space) and how it was rendered (undistorted). This meant that under inspection we were seeing no visual improvement for the cost, with the decision being made to leave out any undistortion steps in the final implementation.

4.8.3 Lucas-Kanade Tracking

As an alternative to the technique of finding features and matching them we tested Lucas-Kinade optical flow [36]. This still requires a step of finding suitable features, for which we used the OpenCV `goodFeaturesToTrack` function, which in turn implements the Shi-Tomasi corner detector [52]. Once the features are found, an image pyramid is generated for both the current and previous frame and the algorithm attempts to relocate those features near to their original position. Again this is provided by OpenCV with the `calcOpticalFlowPyrLK` function. From here the algorithm follows as in our final implementation, by finding a homography between matches with RANSAC then computing the rotation.

A key point to this algorithm is that unlike the brute force matching approach, optical flow is very efficient if features do not move far from their original position but very slow when that is not the case. In our testing it did not take much phone rotation to fall into the latter scenario, much less than we would expect from a casual user. Once frame times begin to increase due to this we run into the downward spiral, and accuracy of any further frames plummets.

4.8.4 FAST and Normalised Cross-Correlation

We also implemented an algorithm based on template matching, in this case Normalised Cross-Correlation (NCC) [32]. In contrast to descriptor matching, template matching compares the direct pixel luminosity values in an area around the feature keypoint with those pixels in a search region in the target image. But it is similar in that we need a set of keypoints to start from — we evaluated Shi-Tomasi corners [52], AGAST [38], and BRISK [31], with the key criterion being speed to find a suitable number of candidate corners. The FAST algorithm [49] proved to be the clear winner, a result that eventually inspired our investigation of ORB.

The application of NCC to feature matching is described by Briechle and Hanebeck [7] — much like optical flow, because a region of pixels is being searched for a match it is a lot more efficient for smaller rotations. In this case we can use the sensor estimate to change the initial search area, in an implementation for real-time panoramas similar to that described by Wagner et al. [60]. Unfortunately, even after taking this step, template matching on 720p images was too demanding for our test phones and we again entered the downward spiral.

4.8.5 Solving Rotation in Sphere-Space

As an alternative to using a homography matrix it is possible to project the found keypoints in each of feature sets $\mathbf{f}_p$ and $\mathbf{f}_c$ to the unit sphere creating vector sets $\mathbf{A}$ and $\mathbf{B}$, then attempt to align the corresponding points with minimal error. To find the rotation result $\mathbf{R}$ in this manner we must solve the Orthogonal Procrustes Problem [22], which we can do using the Kabsch algorithm [24]

$$\mathbf{M} = \mathbf{AB} \tag{4.28}$$

$$\mathbf{U\Sigma V} = \mathrm{SVD}(\mathbf{M}) \tag{4.29}$$

$$\mathbf{R} = \begin{cases} \mathbf{VU}^T & \left|\mathbf{VU}^T\right| = 1 \\ \mathbf{V} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \mathbf{U}^T & \left|\mathbf{VU}^T\right| = -1 \end{cases} \tag{4.30}$$

We used the Eigen library to perform the Singular Value Decomposition (SVD) operation above, and additionally pass the `ComputeThinU` and `ComputeThinV` flags to avoid unnecessary computation.

This approach worked well, but did not provided any performance advantages over the OpenCV RANSAC based homography estimator. We opted to use homographies in our implementation simply because we found them to be conceptually simpler to reason about.

Chapter 5

System Evaluation

Contents

In this chapter we present a both a qualitative and quantitative analysis of various aspects of the environment sharing system we have developed. First we demonstrate the fundamentals of our computer vision algorithm by creating a panorama without the real-time constraint from a pre-recorded video. We then measure the performance profile of each of the SphereMapper and SphereViewer applications, and provide a measure for the end-to-end latency of the entire system during run time. As a measure of quality we export a panorama generated by the application under typical use and inspect the visual quality. Finally we discuss some of the known and potential causes of error found in the created panoramas.

This method of evaluation is consistent with the systems papers discussed in Chapter 2. Due to the novel nature of our system and the complexity of the implementation, we leave a complete user study, including evaluation of usability, presence, and immersion factors, to future work.

For the sending role we evaluate both of our test phones: the LG Nexus 5 and Samsung Galaxy S7, for receiving we only test the S7 as the Nexus is not compatible with the Samsung GearVR mobile HMD. Our goal is to reach real-time performance on both sender and receiver, with a higher framerate for the HMD user, specifically 30 FPS and 60 FPS respectively. We also hope to achieve a panorama quality in our real-time tests that is as close to the offline test as possible. We also show that our new implementation provides an improvement over the similar system PanoVC [40] [60] described in Section 2.4.

5.1 Offline Validation

Our first evaluation is simply to verify that our panorama generation algorithm is capable of producing reasonable quality output — this tests that our method, close to a textbook example [56] but still somewhat modified, does not have any inherent flaws that prevent correct image stitching. It also tests that any quality issues we see during runtime are not caused by simple programming mistakes.

We constructed this test by changing parameters and constants that allow for higher quality stitching without fundamentally changing the process or algorithm. Specifically, we

- use the full sized 9 : 16 FAST feature detector instead of the reduced 7 : 12
- remove the limit of how many found features are stored, and how many feature matches are kept
- increase the accuracy target of the RANSAC process, allowing for more iterations

Figure 4.6 shows a subset of input frames from a video and the resulting panorama generated under these conditions. It shows that our implementation is able to generate a coherent panorama even as the phone camera rotates about all three axes. Notice in particular that the kerb on the left has strong continuity as it crosses all four images, and the horizon is similarly contiguous.

5.2 SphereMapper Performance

Environment sending, as implemented by the SphereMapper application, includes scene capture by the phone camera, pose estimation with computer vision, and sending video frames over the network. In the case of capturing camera frames we cannot precisely

Figure 5.1: This figure demonstrates our rotation estimation and projection algorithms for 4 images each taken 4 video frames apart.

measure the time taken to observe the environment and write the sensor contents to memory; this is because of the event driven nature of the `camera2` API. Whilst it provides an event notifying the end of capture, it does not provide a means to know when the frame capture has started. Likewise we cannot measure the total time WebRTC takes to encode and send a given frame due to an inverse situation — we can measure the start time of frame processing but not the end. Nevertheless, it is possible to measure the average speed of these processes together by substituting our computer vision module with a class that simply immediately returns the identity matrix for our pose, then counting how many frames make it to the encoding stage each second.

Running the application without computer vision yields a constant 30 FPS for capturing, encoding, and network transmission on both the Nexus 5 and Galaxy S7. This is in fact the upper limit of capturing speed for the cameras on each phone, so it is not possible to achieve a higher framerate, implying that there is still some idle CPU time when not using computer vision.

As the computer vision based pose estimation is performed on a separate thread to both the camera capture and video encoding and transmission, the whole process should maintain 30 FPS as long as an estimate is provided within 33ms. Figures 5.2 and 5.3 show the performance profile of our computer vision implementation for the Nexus 5 and Galaxy S7 respectively, broken down into steps as described in Section 4.5.2.

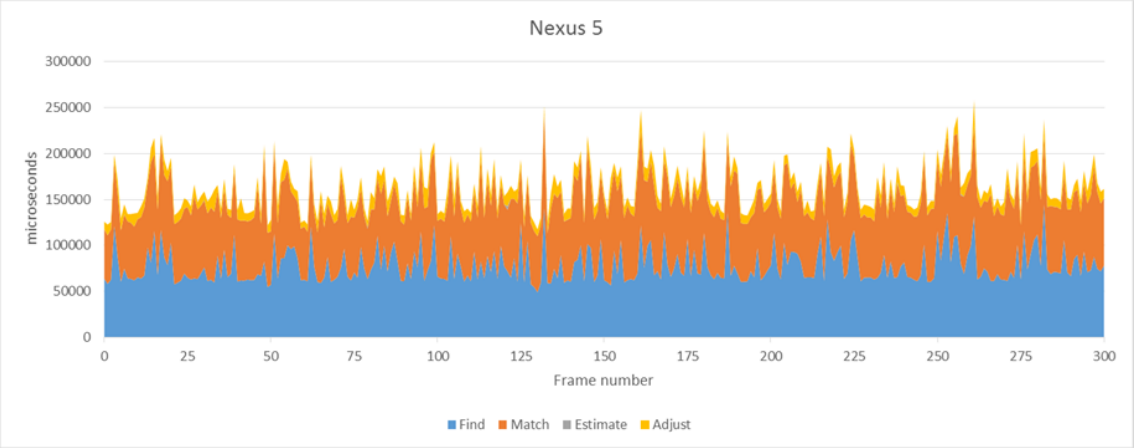

Figure 5.2: Panorama generation performance measurements over 300 frames on the Nexus 5.

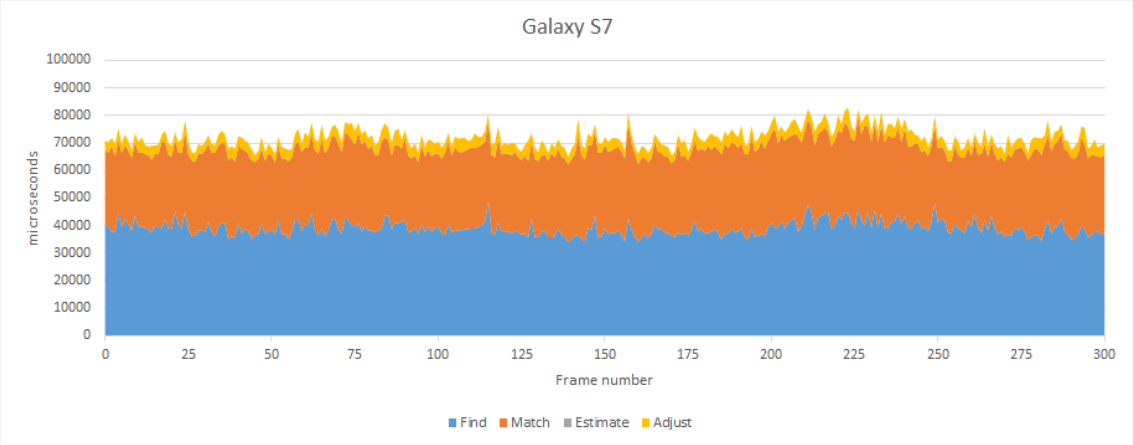

Figure 5.3: Panorama generation performance measurements over 300 frames on the Galaxy S7.

The results show that the computer vision processing time is dominated by the the feature finding and matching tasks. Estimation (homography calculation and RANSAC)

takes a consistent 50 µs, while bundle adjustment takes 3–5ms on the S7 and double that on the Nexus. Of the remaining time, feature finding exhibits the most variance on both phones, though a lot more on the slower Nexus.

As we can see from the charts, our implementation does not reach the performance target on either of our test phones. The Nexus 5 runs at an average speed of 5–7 FPS, whereas the faster Galaxy S7 achieves an average of 14–15 FPS.

5.3 SphereViewer Performance

Viewing the environment with an HMD is implemented in the SphereViewer application and includes receiving and decoding frames from network, reconstructing missing image data from the frame tag, colour space conversion, panorama updating, and eye view projection and rendering. These tasks are split between two separate threads, with everything up to and including live panorama updating on one thread, and final eye view rendering in its own thread, as discussed in Section 4.6.3.

We evaluated the GearVR compatible Galaxy S7 across each thread, in this case measuring the total thread time. In the case of rendering the final output to the display, we consistently measured frame times of under 1ms — well surpassing our target of 16ms. This is not too surprising, as the rendering task consists primarily of simply sampling one texture to paint another, and only for half the screen resolution due to eye duplication.

For the panorama update thread, time from receiving a frame to completely writing it out varied between 2–5ms. This is because, as shown in Figure 4.5, the direction of the camera can have a large effect on how many pixels are processed for a given frame — a side-effect of using an equirectangular projection to store the panorama. Nevertheless, these timings are well under the 33ms threshold required for keeping up with a 30 FPS camera.

One key point to note is that the panorama update rate is limited by the rate of incoming frames, in this case a maximum of 7 FPS from our Nexus 5. Despite this we can still infer that the process would keep up with an incoming stream of 30 FPS based on the timings we observed.

5.4 End-to-end Latency

End-to-end latency refers to the delay in a real-time system between input on one end of a processing pipeline being reflected in the output at the other. In our case this means

the time between either changes in the remote environment or new information captured by rotation the phone being visible to the eye of the HMD user.

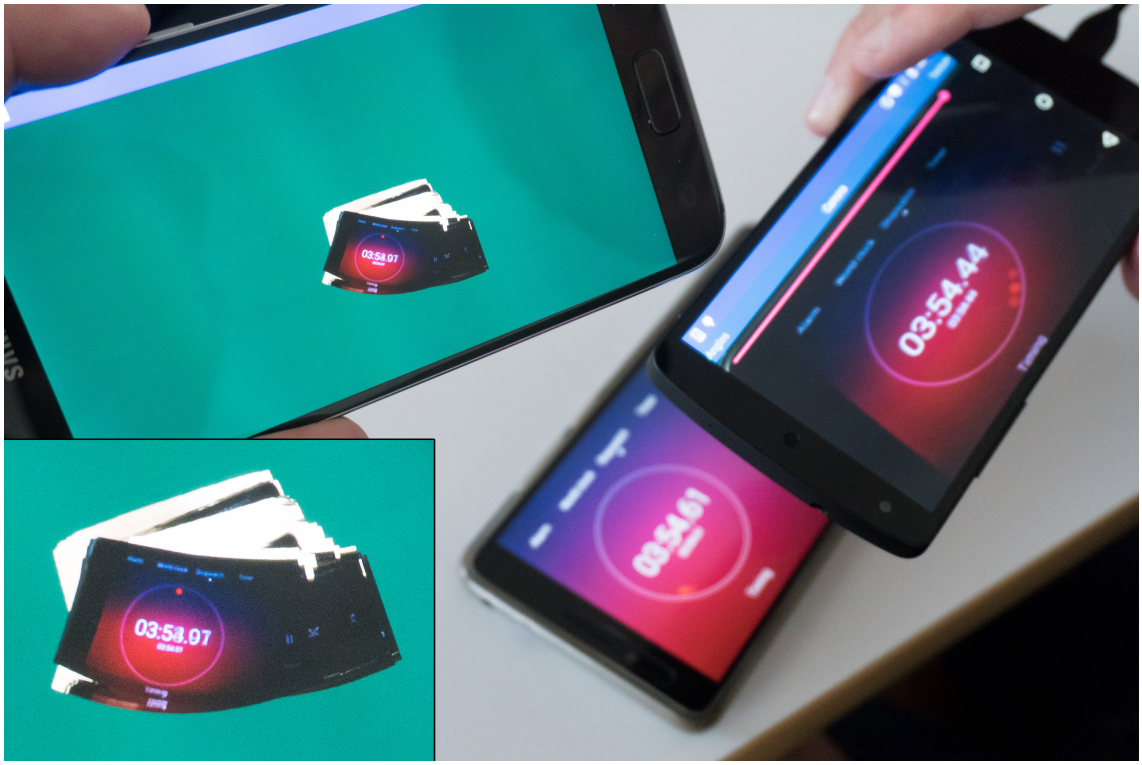

Figure 5.4: An example capture of the end-to-end latecy measurement. The inset shows a closer view of the final output.

To measure end-to-end latency we observe a reference clock with the sending phone, then take a photo with both the reference clock and the receiver display in view. The difference in time shown by the reference and receiver reveals the end-to-end latency. Due to the inability to photograph the receiver display while it is used inside the GearVR, we modified the shader to show the entire panorama buffer regardless of orientation for this test. Additionally, by including the display of the sending phone in the same photo we can determine the intermediate latency at the point of image capture. Figure 5.4 shows one reading taken with this setup, with a reference time of 3:54.61, sender display time of 3:54.44, and panorama update time at 3:54.07. In this case that means a capture latency of less than 170ms, and an overall end-to-end latency of 540ms.

After taking a series of latency measurements we found the average capture latency to be 161ms (or less, due to us not having fine control of the UI thread update) and the overall

latency to average 525ms. As the sending device in this case (the Nexus) can take up to 200ms to process the computer vision, this implies some 300ms worth of communication overhead from encoding, transmission, decoding, and rendering. We view this as a positive outcome, well within the bounds of real-time communication systems.

5.5 Panorama Quality

Our panorama generation is based on 1280×720 (HD) camera frames which have only been processed by an H.264 encoder/decoder before use. This ensures that any single frame when committed to the panorama will be of high quality, so the primary determining factor of the overall panorama's visual quality is the accuracy of the pose estimation algorithm. Poor estimation will manifest as *seams* in sequential frames, showing duplicate or missing rows and columns of pixels near the edge, and as *drift* in frames taken far apart — the scene won't line up as the camera is rotated away then returned to the original position. For the following tests the panoramas were generated starting from the central graffiti wall then moving anti-clockwise in projected panorama space.

Figure 5.5: This panorama was generated in real-time using only the onboard inertial sensors on the Galaxy S7. It has some seam artefacts in the tall building to the left, and visible drift errors to the left of the word 'life'. Note that the final frame, near the bottom left, is of high quality in isolation.

The gyroscope based panorama (Figure 5.5) exhibits one of the key issues of trying to generate a panorama without computer vision: with no way to detect temporary errors, one slight inaccuracy can effect the remainder of the panorama. We can see a clear split in

the scene to the left of the graffiti which changes the horizon level for half the environment. During live testing we also noted the gyroscope alone is quite unstable when the camera is still, leading to visual-motion disconnect for the receiving HMD user.

Figure 5.6: This panorama was generated in real-time using our computer vision method on the Galaxy S7. There are visible seams in the tall building, and along the road above the central wall. Drift is quite limited in this panorama. Note again an area of high quality in the bottom left where the most recent frames have been added.

The computer vision generated panorama shown in Figure 5.6, whilst not as robust as the offline example, shows a coherent, reasonably well stitched panorama. There are seams visible throughout, particularly on the left half of the roadway and on the background building, but there are also areas with accurate stitching such as the river and the close banks. Overall the tracking shows very little drift, with the far bank being nearly perfectly level.

5.6 Causes of Error

There are a number of factors that can cause errors to manifest in the generated environment. The speed at which the user rotates their phone can effect the stitching in a number of ways: at extreme speed we observe motion blur, making it very difficult to find features to track. Slower than that but still fast, there is the possibility of consecutive frames having too little overlap, making for a small set of feature matches to estimate rotation against. Finally, though it does not apply to our brute force matching algorithm, most stitching implementations will become slower as feature pairs move further apart in consecutive frames.

Non-rotational phone movement is also a potential source of error. As noted in Section 3.3, our mathematical model assumes feature to be infinitely far away, which is of course not the case in reality. If the phone camera sensor is positionally moving too much in relation to nearby features in the scene, these mathematical assumptions can break down, leading to poor estimation.

Another source of error is simply too few feature in the scene. Looking at a cloudless sky will cause stitching to be very difficult, if not impossible. Conversely, if a scene has too many features with a repeating pattern or texture, we will have a lot of keypoints but they may each match to multiple different places across frames.

Dynamic elements in the scene can also cause confusion, for example, if a car drives to the right of the camera's view, this could equally be interpreted as a phone rotation to the left, depending on how many other feature matches contradict this. This may be caught by the sensor disagreeing with the vision estimate, but slow moving objects could sneak in under the threshold.

Finally there is a chance of false negatives. Our implementation will drop frames where the computer vision estimate is too different to the sensor reading. In some cases however, it could be the sensors that are incorrect. We take the conservative approach of just stopping processing in case of disagreement, but that means sometimes good estimates are lost.

It is likely that, to a greater or lesser extent, we see all of these error sources manifest in our real-time panorama above.

5.7 Comparison to PanoVC

PanoVC [40] is a system also designed for sharing environments in real-time with panoramas as an intermediate format, based directly on an implementation for generating panoramas on a mobile phone by Wagner et al. [60]. As our conceptual frameworks are quite similar, and PanoVC was developed only last year, we compare the performance of our implementation against PanoVC.

First to consider are the individual frames as captured by the camera, used for computer vision and sent across the network — PanoVC uses 320×240 compared to our 1280×720. Also regarding resolution is the size of the panorama buffer, PanoVC has 2048×512 pixels against our 8192×4096. Together this allows for a much higher resolution for the viewing user with our system.

PanoVC was shown to have an end-to-end latency of approximately 300ms, beating our

implementation by roughly 40%. We also have a direct comparison of tested phones in the Nexus 5: PanoVC runs at 20 FPS compared to our 7 FPS. This is perhaps unsurprising given the vast difference in data size handled.

Given these comparisons we believe we have made a significant improvement compared to what is the existing state of the art implementation. We did in fact implement the technique used by PanoVC during our investigation phase – template matching with normalised cross-correlation (see Section 4.8.4) – and found it to be too slow for the resolution of imagery we are targeting.

5.8 Summary

Regarding overall panorama quality, we can see there is still some improvement to make. The offline verification shows that the fundamentals are sound — future work could involve further exploration of the trade-offs between speed and accuracy. Nevertheless, from our time testing the system ourselves, the HMD user can get a good sense of the remote environment from the panoramas we generate.

Some papers [23] [40] [58] define real-time as 15 FPS, which we do achieve on the faster of the phones we tested, but based on developer guidelines* we consider this too slow for applications involving head-mounted displays. We did however achieve our target rendering speed of 60 FPS on the head-mounted display, meaning the user will not have their immersion and sense of presence affected by local performance issues. The latency of the system is low enough for real-time communication without any issue.

*`https://developer3.oculus.com/documentation/publish/latest/concepts/publish-mobile-req/`

Chapter 6

Summary and Conclusion

In this thesis we have presented a novel system for enabling mobile head-mounted display users to be virtually transported to remote locations, facilitated by our new applications SphereMapper and SphereViewer, as part of an *immersive panoramic call*. Our goal is to create the experience of telepresence and copresence within an HMD user, so they can feel 'really there' with a partner in a remote environment. We outline some of the key findings from undertaking this work below.

6.1 Summary of Findings

We successfully developed a prototype that allowed us to carry out immersive panoramic calls. We have pushed mobile phone performance to create robust panoramas in real-time, allowing for the sharing of arbitrary distant environments with HMD users. We have outlined a framework for updating 360° environments in real-time, and demonstrated that is possible through a working prototype.

Our system was split up across two applications, one for each of the mobile phone user sending the environment and one for the HMD user viewing it. We reached our performance targets for the HMD based app, but had only moderate success on the sending side, owing to our commitment to creating high resolution panoramas. That being said, our implementation may be the fastest high resolution panorama stitcher developed for mobile phones to date.

During the course of our implementation we made an open source contribution to the OpenCV library which has already been accepted, and we have developed a WebRTC module that will be prepared for submission at the conclusion of this work. The system

is also already being used as a research platform by another student — he is testing novel ways to interact when wearing a head-mounted display.

6.2 Future Work

This platform we have developed offers many potential avenues for future research, from technical issues such as panorama quality and processing speed, to user interactions and the study of presence. It is also in a state where it could be adapted to commercial usage such as remote assistance, virtual tourism, or just as a general communication tool.

6.2.1 Performance and Accuracy

We believe that, whilst we have put significant effort into creating applications that run as fast as possible, there is still many improvements to be made in regards to performance and accuracy. These two factors will often go together, as improving in one area may allow time to focus on the other. As more powerful devices become available with each passing year, an interesting research route would be adaptive algorithms that are able to scale up or down based on the computation power available.

6.2.2 User Interaction

The user interaction model in immersive panoramic calls could be extended in a number a ways. A simple feature to add which may improve copresence would be an overlay showing each user where the other participant is looking — this would allow the users to align on areas of interest and keep the live update area where the HMD user wants it.

An existing project already under research by another student is the ability to cut imagery and hands out of the frame before updating the panorama — users can then point out into the scene without worrying about affecting the computer vision stitching or leaving an imprint of their hand in the panorama.

There is also an interesting avenue to explore in swapping environments — when the GearVR is in use it is still possible to access the rear camera on the phone; we could use this fact to allow two HMD users to completely swap locations with one another.

6.2.3 Presence User Study

Based on related work, we have speculated that our system would improve the sense of presence for at least the HMD user in a call, and possibly both participants. The next

step then is to put this hypothesis to the test with a user study where participants must reason about the remote environment from within an HMD. We would also combine this with other participatory studies such as system usability.

Bibliography

[1] Adams, A., Gelfand, N., and Pulli, K. (2008). Viewfinder alignment. In *Computer Graphics Forum*, volume 27, pages 597–606. Wiley Online Library.

[2] Agarwala, A., Zheng, K. C., Pal, C., Agrawala, M., Cohen, M., Curless, B., Salesin, D., and Szeliski, R. (2005). Panoramic video textures. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 821–827. ACM.

[3] Allison, R. S., Gillam, B. J., and Vecellio, E. (2009). Binocular depth discrimination and estimation beyond interaction space. *Journal of Vision*, 9(1):10–10.

[4] Baudisch, P., Tan, D., Steedly, D., Rudolph, E., Uyttendaele, M., Pal, C., and Szeliski, R. (2005). Panoramic viewfinder: providing a real-time preview to help users avoid flaws in panoramic pictures. In *Proceedings of the 17th Australia conference on Computer-Human Interaction: Citizens Online: Considerations for Today and the Future*, pages 1–10. Computer-Human Interaction Special Interest Group (CHISIG) of Australia.

[5] Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer.

[6] Biocca, F. and Delaney, B. (1995). Immersive virtual reality technology. *Communication in the age of virtual reality*, pages 57–124.

[7] Briechle, K. and Hanebeck, U. D. (2001). Template matching using fast normalized cross correlation. In *Aerospace/Defense Sensing, Simulation, and Controls*, pages 95–102. International Society for Optics and Photonics.

[8] Cadena, C., Carlone, L., Carrillo, H., Latif, Y., Scaramuzza, D., Neira, J., Reid, I., and Leonard, J. J. (2016). Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332.

[9] Calonder, M., Lepetit, V., Strecha, C., and Fua, P. (2010). Brief: Binary robust independent elementary features. In *European conference on computer vision*, pages 778–792. Springer.

[10] Dalvandi, A., Riecke, B. E., and Calvert, T. (2011). Panoramic video techniques for improving presence in virtual environments. In *Proceedings of the 17th Eurographics*

conference on Virtual Environments & Third Joint Virtual Reality, pages 103–110. Eurographics Association.

[11] DiVerdi, S., Wither, J., and Hollerer, T. (2008). Envisor: Online environment map construction for mixed reality. In *Virtual Reality Conference, 2008. VR'08. IEEE*, pages 19–26. IEEE.

[12] Drugge, M., Nilsson, M., Parviainen, R., and Parnes, P. (2004). Experiences of using wearable computers for ambient telepresence and remote interaction. In *Proceedings of the 2004 ACM SIGMM workshop on Effective telepresence*, pages 2–11. ACM.

[13] Durlach, N. and Slater, M. (2000). Presence in shared virtual environments and virtual togetherness. *Presence: Teleoperators and Virtual Environments*, 9(2):214–217.

[14] Fischler, M. A. and Bolles, R. C. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395.

[15] Fitzgibbon, A. W. (2001). Stochastic rigidity: Image registration for nowhere-static scenes. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 1, pages 662–669. IEEE.

[16] Gauglitz, S., Nuernberger, B., Turk, M., and Höllerer, T. (2014). World-stabilized annotations and virtual scene navigation for remote collaboration. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 449–459. ACM.

[17] Google – Android Developers (2017). Sensors Overview. `https://developer.android.com/guide/topics/sensors/sensors_overview.html`. [Online; accessed 26 January 2017].

[18] Handley, M., Perkins, C., and Jacobson, V. (2006). Sdp: session description protocol.

[19] Hartley, R. and Zisserman, A. (2003). *Multiple view geometry in computer vision*. Cambridge university press.

[20] Horvath, K. and Lombard, M. (2010). Social and spatial presence: An application to optimize human-computer interaction. *PsychNology Journal*, 8(1):85–114.

[21] Howarth, P. and Costello, P. (1997). The occurrence of virtual simulation sickness symptoms when an hmd was used as a personal viewing system. *Displays*, 18(2):107–116.

[22] Hurley, J. R. and Cattell, R. B. (1962). The procrustes program: Producing direct rotation to test a hypothesized factor structure. *Systems Research and Behavioral Science*, 7(2):258–262.

[23] Jo, H. and Hwang, S. (2013). Chili: viewpoint control and on-video drawing for mobile video calls. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 1425–1430. ACM.

[24] Kabsch, W. (1976). A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A: Crystal Physics, Diffraction, Theoretical and General Crystallography*, 32(5):922–923.

[25] Kasahara, S., Nagai, S., and Rekimoto, J. (2015). First person omnidirectional video: System design and implications for immersive experience. In *Proceedings of the ACM International Conference on Interactive Experiences for TV and Online Video*, pages 33–42. ACM.

[26] Kasahara, S. and Rekimoto, J. (2014). Jackin: integrating first-person view with out-of-body vision generation for human-human augmentation. In *Proceedings of the 5th Augmented Human International Conference*, page 46. ACM.

[27] Kasahara, S. and Rekimoto, J. (2015). Jackin head: immersive visual telepresence system with omnidirectional wearable camera for remote collaboration. In *Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology*, pages 217–225. ACM.

[28] Kratz, S., Kimber, D., Su, W., Gordon, G., and Severns, D. (2014). Polly: Being there through the parrot and a guide. In *Proceedings of the 16th international conference on Human-computer interaction with mobile devices & services*, pages 625–630. ACM.

[29] LaValle, S. M., Yershova, A., Katsev, M., and Antonov, M. (2014). Head tracking for the oculus rift. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 187–194. IEEE.

[30] Lee, K. M. (2004). Presence, explicated. *Communication theory*, 14(1):27–50.

[31] Leutenegger, S., Chli, M., and Siegwart, R. Y. (2011). Brisk: Binary robust invariant scalable keypoints. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2548–2555. IEEE.

[32] Lewis, J. (1995). Fast normalized cross-correlation. In *Vision interface*, volume 10, pages 120–123.

[33] Lewis, J. R. and Sauro, J. (2009). The factor structure of the system usability scale. In *International Conference on Human Centered Design*, pages 94–103. Springer.

[34] Lombard, M. and Ditton, T. (1997). At the heart of it all: The concept of presence. *Journal of Computer-Mediated Communication*, 3(2).

[35] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110.

[36] Lucas, B. D. and Kanade, T. (1981). An iterative image registration technique with an application to stereo vision.

[37] Mahy, R., Matthews, P., and Rosenberg, J. (2010). Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). Technical report.

[38] Mair, E., Hager, G. D., Burschka, D., Suppa, M., and Hirzinger, G. (2010). Adaptive and generic corner detection based on the accelerated segment test. In *European conference on Computer vision*, pages 183–196. Springer.

[39] Mozilla Developer Network (2016). WebRTC connectivity – Web APIs. `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity`. [Online; accessed 26 January 2017].

[40] Müller, J., Langlotz, T., and Regenbrecht, H. (2016). Panovc: Pervasive telepresence using mobile phones. In *Pervasive Computing and Communications (PerCom), 2016 IEEE International Conference on*, pages 1–10. IEEE.

[41] Nayar, S. K. (1997). Catadioptric omnidirectional camera. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 482–488. IEEE.

[42] Neumann, U., Pintaric, T., and Rizzo, A. (2000). Immersive panoramic video. In *Proceedings of the eighth ACM international conference on Multimedia*, pages 493–494. ACM.

[43] Pacha, A. (2013). Sensor fusion for robust outdoor augmented reality tracking on mobile devices.

[44] Pavlenko, Andrey (2017). About – OpenCV Library. `http://opencv.org/about.html`. [Online; accessed 30 March 2017].

[45] Pece, F., Steptoe, W., Wanner, F., Julier, S., Weyrich, T., Kautz, J., and Steed, A. (2013). Panoinserts: mobile spatial teleconferencing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1319–1328. ACM.

[46] Pece, F., Tompkin, J., Pfister, H., Kautz, J., and Theobalt, C. (2014). Device effect on panoramic video+ context tasks. In *Proceedings of the 11th European Conference on Visual Media Production*, page 14. ACM.

[47] Rae, I., Venolia, G., Tang, J. C., and Molnar, D. (2015). A framework for understanding and designing telepresence. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 1552–1566. ACM.

[48] Rosenberg, J. (2010). Interactive connectivity establishment (ice): A methodology for network address translator (nat) traversal for offer/answer protocols.

[49] Rosten, E. and Drummond, T. (2006). Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer.

[50] Rublee, E., Rabaud, V., Konolige, K., and Bradski, G. (2011). Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE.

[51] Schubert, T., Friedmann, F., and Regenbrecht, H. (2001). The experience of presence: Factor analytic insights. *Presence: Teleoperators and virtual environments*, 10(3):266–281.

[52] Shi, J. et al. (1994). Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE.

[53] Srisuresh, P. and Holdrege, M. (1999). Ip network address translator (nat) terminology and considerations.

[54] Sutherland, I. E. (1968). A head-mounted three dimensional display. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 757–764. ACM.

[55] Szeliski, R. (1996). Video mosaics for virtual environments. *IEEE computer Graphics and Applications*, 16(2):22–30.

[56] Szeliski, R. (2006). Image alignment and stitching: A tutorial. *Foundations and Trends in Computer Graphics and Vision*, 2(1):1–104.

[57] Szeliski, R. and Shum, H.-Y. (1997). Creating full view panoramic image mosaics and environment maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 251–258. ACM Press/Addison-Wesley Publishing Co.

[58] Tang, W.-K., Wong, T.-T., and Heng, P.-A. (2005). A system for real-time panorama generation and display in tele-immersive applications. *IEEE Transactions on Multimedia*, 7(2):280–292.

[59] Valin, J.-M., Vos, K., and Terriberry, T. (2012). Definition of the opus audio codec. Technical report.

[60] Wagner, D., Mulloni, A., Langlotz, T., and Schmalstieg, D. (2010). Real-time panoramic mapping and tracking on mobile phones. In *Virtual Reality Conference (VR), 2010 IEEE*, pages 211–218. IEEE.

[61] Wiegand, T., Sullivan, G. J., Bjontegaard, G., and Luthra, A. (2003). Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576.

[62] Wing, D., Matthews, P., Mahy, R., and Rosenberg, J. (2008). Session traversal utilities for nat (stun).

[63] Yang, R., Kurashima, C. S., Towles, H., Nashel, A., and Zuffo, M. K. (2007). Immersive video teleconferencing with user-steerable views. *Presence*, 16(2):188–205.

[64] Zhang, Z. (2000). A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22(11):1330–1334.